

Institut für Softwaretechnologie

Universität Stuttgart
Universitätsstraße 38
D-70569 Stuttgart

Bachelorarbeit

Automatische Analyse der Wartbarkeit von Services anhand der OpenAPI Spezifikation

Kai Chen

Studiengang: Softwaretechnik
Prüfer/in: Prof. Dr. Stefan Wagner
Betreuer/in: Justus Bogner

Beginn am: 16. April 2018
Beendet am: 16. Oktober 2018

Kurzfassung

RESTful Webservices gewinnen in letzter Zeit immer mehr an Bedeutung und ersetzen traditionelle Webservices mit SOAP und WDSL. Im Rahmen der sich immer schneller verändernden Umwelt und damit einhergehenden wechselnden Anforderungen ist es für Unternehmen überlebenswichtig, anpassbare Services und damit auch flexible Schnittstellen zu haben. Ein wichtiges Qualitätsattribut für Webservices ist die Wartbarkeit, die in diesem Kontext eine besondere Bedeutung hat. Um einschätzen zu können, wie gut wartbar eine RESTful API ist, wird in dieser Arbeit ein prototypisches System zur automatischen Analyse von RESTful APIs mithilfe der OpenAPI Spezifikation entwickelt. Das System beinhaltet zwei Komponenten: eine Kernkomponente, die über die Kommandozeile ausgeführt wird und eine Webapplikation, die die Funktionalitäten des Systems als RESTful Webservice anbietet. Zur Bewertung der Wartbarkeit werden exemplarisch die Metriken *Service Interface Data Cohesion* und *Weighted Service Interface Count* benutzt. Der hier entwickelte Prototyp dient zur Präsentation des Tools und dessen Einsatz, und wurde daher mit dem Fokus auf einfache Erweiterbarkeit konzipiert.

Inhaltsverzeichnis

1	Einleitung	9
1.1	Motivation	9
1.2	Zielsetzung und Aufbau der Arbeit	9
2	Technischer Hintergrund	11
2.1	Service-orientierte Systeme	11
2.2	REpresentational State Transfer (REST)	18
2.3	Swagger/OpenAPI	24
2.4	Wartbarkeit	29
3	Verwandte Arbeiten	37
3.1	Analyse von SOAP-basierten Webservices	37
3.2	Analyse von RESTful APIs	40
4	Entwicklung eines OpenAPI Evaluationstools	43
4.1	Anforderungen	43
4.2	Methodisches Vorgehen	44
4.3	Architektur	47
4.4	Anwendungsbeispiel	53
5	Fazit	59
5.1	Limitationen der Arbeit	59
5.2	Ausblick	60
	Literaturverzeichnis	63

Abbildungsverzeichnis

2.1	Struktur der OpenAPI Spezifikationen Version 2 und 3 [OAI17]	27
3.1	Überblick der untersuchten Metriken [HLV18]	40
4.1	Übersicht über das Gesamtprojekt	47
4.2	Komponentendiagramm	48
4.3	Paketdiagramm Kommandozeilenprogramm	50
4.4	Paketdiagramm Webapplikation	51
4.5	Datenbankdiagramm	53
4.6	Use Case-Diagramm Kommandozeilenapplikation	53
4.7	Use Case-Diagramm Webapplikation	55

1 Einleitung

Einer der Begriffe, die in den letzten Jahren immer mehr an Bedeutung gewonnen haben, ist "RESTful Webservices". Sie lösen die klassischen Webservices über SOAP und WSDL ab und gewinnen im Rahmen des Wachstums des Internet of Things und der Microservices noch mehr an Bedeutung. Allerdings fällt auf, dass viele RESTful Webservices nicht REST-konform sind. Daher ist es sinnvoll, sich mit diesem Thema genauer auseinanderzusetzen.

1.1 Motivation

RESTful Webservices kommunizieren über Schnittstellen miteinander, die RESTful APIs. Damit sich Unternehmen in der sich immer schneller verändernden Unternehmensumwelt an neue Anforderungen anpassen können, ist es notwendig, dass die Schnittstellen ihrer Webservices flexibel sind und verändert werden können. So können die Services auf neue Art und Weise miteinander verbunden werden und die Konkurrenzfähigkeit des Unternehmens sicherstellen.

Um diese Flexibilität der Schnittstellen zu erreichen, ist eine gute Wartbarkeit der API notwendig. Die Wartung ist außerdem der teuerste Abschnitt im Softwarelebenszyklus und sollte daher schon zu Beginn der Entwicklung berücksichtigt werden. Diese Tatsachen machen die Wartbarkeit zu einem sehr wichtigen Qualitätsattribut von Softwareprojekten, zu denen auch Webservices zählen.

Es existieren Guidelines, die die Qualität von RESTful APIs analysieren, jedoch sind diese oft subjektiv und manuell auszuführen. Durch die Einführung der API Beschreibungssprache *OpenAPI Spezifikation* ist es nun möglich, automatische und objektive Analysen von RESTful APIs anhand der Struktur der Spezifikation durchzuführen. Wie jedes Softwareartefakt hat auch die OpenAPI Spezifikation Attribute, die auf die Wartbarkeit geprüft werden können. Für diese Analyse sind Metriken notwendig, die auf die Spezifikation angewendet werden und dadurch Informationen über die Qualität der Schnittstelle generieren.

1.2 Zielsetzung und Aufbau der Arbeit

Eine solche automatische Analyse der Wartbarkeit von RESTful APIs deckt Mängel auf, kann dadurch die Qualität der Schnittstellen erhöhen und somit dem Unternehmen mehr Flexibilität und gleichzeitig Sicherheit und Zukunftsfähigkeit ermöglichen.

Im Rahmen dieser Arbeit wird ein prototypisches Softwaresystem zur Analyse von RESTful APIs anhand der OpenAPI Spezifikation entwickelt. Der Fokus des Projekts liegt auf dem Ansatz und der Architektur des Systems und weniger auf der Vollständigkeit. Aus diesem Grund soll das System in der Zukunft einfach zu erweitern sein. Zu Demonstrationszwecken werden beispielhaft zwei für die Wartbarkeit relevante Metriken implementiert.

Die Arbeit ist in folgender Weise gegliedert: Zu Beginn werden in Kapitel 2 die theoretischen Grundlagen genauer beschrieben. Zuerst werden Service-orientierte Systeme erläutert, daraufhin wird speziell auf REST Services eingegangen. Anschließend wird die OpenAPI Spezifikation dargestellt und zum Schluss werden im Unterkapitel Wartbarkeit dieses Qualitätsattribut und die entsprechenden Metriken vorgestellt. Nachdem die Grundlagen vorhanden sind, werden in Kapitel 3 verwandte Arbeiten aufgezeigt. Hierbei handelt es sich um wissenschaftliche Arbeiten, die ebenfalls Analysen mithilfe von Metriken durchgeführt haben oder in anderer Weise für das Projekt relevant sind und als Inspiration dienen. Mit all diesem Vorwissen geht es dann in Kapitel 4 an die Beschreibung des eigenen Softwareprojekts, der Hauptaufgabe dieser Arbeit. In diesem Kapitel werden die Anforderungen aufgezeigt, das Vorgehen und die Architektur des Systems erläutert und anhand eines Anwendungsbeispiels verdeutlicht. Zum Abschluss folgt in Kapitel 5 eine kurze Zusammenfassung sowie eine kritische Würdigung der Arbeit und ein Ausblick auf mögliche Erweiterungen und zukünftige Erkenntnisse.

2 Technischer Hintergrund

In diesem Kapitel wird das theoretische Wissen vermittelt, das benötigt wird, um die Wartbarkeit von RESTful APIs anhand der OpenAPI Spezifikation zu analysieren. Es besteht aus den Unterkapiteln Service-orientierte Systeme, REpresentational State Transfer, Swagger/OpenAPI und Wartbarkeit.

2.1 Service-orientierte Systeme

Obwohl der Begriff der Service-orientierten Architektur (SOA) bereits seit 1996 existiert, gibt es immer noch keine eindeutige, überall akzeptierte Definition. Oft fehlen in einer Definition wichtige Aspekte, die in einer anderen besonders hervorgehoben werden. Besonders häufig wird jedoch die Definition der Oasis Gruppe benutzt:

„SOA ist ein Paradigma für die Strukturierung und Nutzung verteilter Funktionalität, die von unterschiedlichen Besitzern verantwortet wird.“ [MLM+06]

Im Allgemeinen sagt Service-orientierte Architektur aus, dass die technischen Elemente der Anwendungsentwicklung wie z.B. der Programmcode zu höherwertigen Komponenten zusammengefasst werden, die fachlich zusammengehören. Diese zusammengehörenden SOA-Komponenten bieten sogenannte Dienste an, welche benutzt werden können, um bestimmte Business Probleme zu lösen. Dienste werden für potentielle Nutzer sichtbar gemacht und können mit diesen durch Informationsaustausch interagieren. [NS11]

Laskey et al. [LL09] charakterisieren auf SOA basierende Systeme wie folgt: Der erste wichtige Punkt ist die Sichtbarkeit der Bedürfnisse, also die Fähigkeit, dass Verbraucher mit bestimmten Bedürfnissen leicht diejenigen Anbieter finden können, die ihre Bedürfnisse optimal abdecken. Des Weiteren muss ein Mittel zur Interaktion zwischen Verbrauchern und Anbietern gegeben sein. Außerdem soll die Anwendung Geschäftsprobleme der Nutzer adressieren und hat somit Auswirkungen auf die reale Welt.

2.1.1 Merkmale von SOA

Eines der Hauptprinzipien von Service-orientierten Architekturen ist die lose Kopplung zwischen verschiedenen Diensten. Die Kopplung ist der Grad, in dem eine Einheit, z.B. ein Client, von einer anderen Einheit abhängt. Eine Kopplung wird als eng eingestuft, wenn starke Abhängigkeiten zwischen Diensten existieren. Eine enge Kopplung zwischen zwei Einheiten führt dazu, dass bei Änderungen einer Einheit gegebenenfalls Anpassungen an der anderen Einheit durchgeführt werden müssen. Wenn die existierenden Abhängigkeiten eher schwach sind, wird die Kopplung

als lose bezeichnet. [Dai11] Zum Zeitpunkt der Kompilierung eines SOA Programms ist oft nicht bekannt, welcher Dienst zur Laufzeit aufgerufen wird. Die Dienste werden erst bei Bedarf dynamisch eingebunden, was eine lose Kopplung charakterisiert. [Mel10]

Neben der losen Kopplung wurden von Erl et al. [EGN+14] weitere Service-orientierte Designprinzipien präsentiert:

- *Standardisierter Service Vertrag*: Dienste vom selben Servicebestand entsprechen dem selben Vertragsdesign-Standard.
- *Service Abstraktion*: Serviceverträge enthalten nur essentielle Informationen, also solche, die für die Nutzung des Services notwendig sind.
- *Service Wiederverwendbarkeit*: Dienste enthalten und formulieren agnostische Logiken und sind wiederverwendbare Unternehmensressourcen.
- *Service Autonomie*: Dienste haben ein hohes Maß an Kontrolle über die zugrunde liegende Laufzeitumgebung.
- *Service Zustandslosigkeit*: Services minimieren den Ressourcenverbrauch, indem sie die Verwaltung von Zustandsinformationen bei Bedarf verschieben.
- *Service Entdeckbarkeit*: Dienste werden durch kommunikative Metadaten ergänzt, mit denen sie effektiv entdeckt und interpretiert werden können.
- *Service Kombinierbarkeit*: Services sind effektive Kompositionsteilnehmer, unabhängig von der Größe und Komplexität der Komposition.

Ein Merkmal von SOA ist die Existenz von globalen Registries und Repositories, in denen vorhandene Dienste gespeichert werden. Hier kann nach Methoden gesucht werden, die für die spezifische Anwendung benötigt werden. Registry und Repository unterscheiden sich dadurch, dass im Repository die Software für die Dienste und zugehörige Metadaten gespeichert werden. Die Registry hingegen ist lediglich ein Katalog mit Verweisen auf die Metadaten der verfügbaren Services. [Tec18] Die Interaktion mit dem dort gefundenen Dienst kann mit einer maschinenlesbaren Schnittstelle verwirklicht werden. Diese soll offene Standards benutzen, sodass eine breite Akzeptanz der Architektur sichergestellt wird.

Notwendige Voraussetzungen für eine SOA sind außerdem Einfachheit, Sicherheit und Akzeptanz. Einfachheit fördert eine schnelle Umsetzung, Sicherheit ist generell ein wichtiges Thema in der IT, insbesondere auch in Bezug auf Webservices. Die Akzeptanz einer SOA ist bedingt durch diese beiden Punkte. Weiterhin ist zu beachten, dass die Kommunikation zwischen den Services vollkommen automatisiert abläuft. Der Mensch hat keinen Einfluss darauf.

Dadurch, dass SOAs flexibel und lose gekoppelt sind, sind sie optimal geeignet zur Implementierung von Geschäftsprozessmodellen. Diese werden oft von externen Ereignissen beeinflusst, weshalb Komponenten einer SOA in der Lage sein sollen, richtig und schnell auf diese Ereignisse zu reagieren. Aus diesem Grund werden solche Architekturen auch als „Ereignisgetriebene Architekturen“ bezeichnet. [Mel10]

Erl et al. [EGN+14] führten in ihrer Arbeit vier grundlegende Charakteristika von SOA auf:

- SOAs sind geschäftsgetrieben: Das heißt, die Architektur des Systems ist an der Geschäftsarchitektur ausgerichtet. Sie verändert sich mit dieser und passt sich ihr an, unterstützt Veränderungen der Geschäftsprozesse und ist somit immer auf das Business fokussiert.
- SOAs sind anbieterunabhängig: Das Architekturmodell ist nicht von einer Plattform abhängig, es können verschiedene Anbieter genutzt werden. So können Technologien über die Zeit hinweg ersetzt oder miteinander kombiniert werden, um die Geschäftsanforderungen jederzeit bestmöglich erfüllen zu können.
- SOAs sind unternehmens-zentriert: Die Architektur repräsentiert einen bedeutenden Teil des Unternehmens, indem sie die Wiederverwendbarkeit und Zusammensetzung von Services und serviceorientierten Lösungen ermöglicht und damit die traditionellen Applikationssilos ersetzt.
- SOAs sind kombinations-zentriert: Die Architektur ermöglicht eine agile Änderung der Zusammensetzung der Services und kann sich dadurch Veränderungen jederzeit anpassen.

2.1.2 Komponenten von SOA

Eine SOA-Komponente bietet viele öffentliche Schnittstellen in Form von Diensten mit fachlichen Funktionen an. Diese Schnittstellen können zum Beispiel in Form von REpresentational State Transfer (REST), Webservices Description Language (WSDL) oder Remote Procedure Call (RPC) implementiert werden. Standardmäßig basiert die Kommunikation auf den Prinzipien Anfrage/Antwort, Anfrage/Reaktion oder Publish/Subscribe.

Die öffentlichen Eigenschaften einer Komponente und ihrer Dienste werden in einem sogenannten Servicevertrag definiert. Dieser beschreibt die Metadaten der jeweiligen Komponente, wie beispielsweise Basisdaten, Ownership oder fachliche und technische Definitionen der Schnittstellen. Außerdem können Dienste gleicher Art in Schichten gruppiert werden, wodurch die Komplexität sinkt. Beispiele von Schichten sind: Frontend, Prozesskomponenten, Orchestrierungskomponenten und Basiskomponenten. [NS11] Dabei gilt:

- Frontend ist die Ebene, die Interaktionen mit den Elementen der Architektur ermöglicht z.B. in Form einer grafischen Benutzeroberfläche.
- Prozesskomponenten steuern den Prozessablauf, beinhalten Interaktionen mit dem Nutzer und dauern oftmals sehr lange an.
- Orchestrierungskomponenten können Kombinationen von Basis-Diensten schaffen oder technische Integrationsprobleme lösen. Sie beinhalten keine fachliche Logik.
- Basiskomponenten verwalten den Lebenszyklus der Objekte. Sie können benutzt werden, um Business Objekte zu erzeugen, zu suchen, zu ändern oder zu löschen.

Wichtig hierbei ist, dass sowohl SOA-Komponenten als auch Schichten komplett technologieunabhängig sind, d.h. sie sind Blackboxen, bei denen lediglich die Schnittstellen verbindlich sind. Die eigentlichen Implementierungsdetails sollen dabei versteckt bleiben.

Dienste/Services

Ein Dienst, auch Service genannt, ist eine Softwarekomponente, die über eine Schnittstelle von anderen Nutzern verwendet werden kann. Oftmals werden die Dienste mit typischen internetfähigen Sprachen und Protokollen implementiert und daher Webservices genannt. [MZM+17] Die Details der Technologie und Implementierung der Schnittstelle sind dabei für die Außenwelt nicht sichtbar. Daher muss eine sogenannte Service Description in maschinenlesbarer Form vorhanden sein, damit beispielsweise andere Dienste darauf zugreifen können. Auf den Dienst kann nur über diese Schnittstelle zugegriffen werden. Eine Service Description ist zum Beispiel die „Web Services Description Language (WSDL)“.

Damit ein Dienst von anderen Nutzern gefunden und benutzt werden kann, muss der Dienstanbieter seinen Dienst in einem Dienstverzeichnis registrieren. Dabei ist der Dienstanbieter auch für die Verfügbarkeit des Dienstes zuständig. Das heißt er muss sich um dessen Betrieb und Wartung kümmern, und dafür sorgen, dass die Plattform, auf der der Dienst angeboten wird, aufrechterhalten wird. Außerdem ist er für die Sicherheit seiner Plattform zuständig. Der Dienstanbieter muss nicht alle angebotenen Dienste selbst programmieren, sondern kann andere Dienste aus dem Netz holen und diese auch, falls nötig, miteinander kombinieren.

Das Dienstverzeichnis ist für das Finden des Dienstes zuständig. Es ist möglich, einen Dienst zu entwickeln, der als Suchmaschine für andere Dienste fungiert. Dabei ist wichtig, dass die Dienste, die registriert werden, eindeutig einer Kategorie zugeordnet werden können.

Damit die Interaktion mit dem Dienst gut funktioniert, müssen folgende Aktionen durchgeführt werden [Me10]:

- Eintragen des Dienstes in ein Verzeichnis
- Installation des Dienstes in einer entsprechenden Umgebung
- Erstellung einer treffenden Beschreibung für die Suche
- Suche nach dem Dienst
- Abfrage der Schnittstellenbeschreibung des Dienstes
- Abfrage der Voraussetzungen zur Nutzung des Dienstes, z.B. Zertifikat

2.1.3 Microservice-orientierte Architektur

Mit der Zeit haben Unternehmen gelernt, dass SOA ein großer, kostspieliger und komplizierter Architekturstil ist, bei dem das Designen und Implementieren der Software sehr viel Zeit in Anspruch nimmt. Eine vielversprechende Alternative, die diese Probleme adressiert, ist die Microservice-Architektur. Mit Microservices ist ein neuer Architekturstil zum Entwickeln von hochskalierbaren und modularen Anwendungen entstanden.

Ein Microservice ist eine kleine Software-Komponente, die nur eine einzelne Aufgabe hat und unabhängig von anderen Microservices deployed, skaliert und getestet werden kann. Solch eine einzelne Aufgabe kann sowohl funktional, als auch nicht-funktional oder funktionsübergreifend sein. [Thö15] Microservice-orientierte Systeme sind daher Systeme, die aus einer Reihe kleiner Services bestehen, die jeweils in einem eigenen Prozess laufen und über klar definierte APIs miteinander

kommunizieren. Dazu werden Protokolle benutzt die eine lose Kopplung unterstützen, wie zum Beispiel REST. [FL15] Die Microservices können eigene Unterstützungsdienste wie beispielsweise eine spezielle Datenbank mitbringen. [Wol18]

Die Charakteristika von Microservices sind nach [Sma18] und [FL15] folgende:

- Eine Anwendung einer Microservice-orientierten Architektur besteht aus vielen individuellen Komponenten, die unabhängig voneinander austauschbar und erweiterbar sind. Ein Vorteil der Verwendung von Komponenten ist, dass nicht nach jeder Änderung die gesamte Anwendung neu deployed werden muss, sondern es reicht oft aus, den einen veränderten Service neu zu deployen.
- Ein Microservice ist typischerweise auf Unternehmen ausgerichtet. Im Gegensatz zur traditionellen Softwareentwicklung, bei der die Software nach dem Projekt als fertig angesehen und an ein anderes Team abgegeben wird, geht es hier um eine kontinuierliche Beziehung zwischen Projekt und Team. Die Teams sind funktionsübergreifend für den Service zuständig, und nicht nur für einzelne Teilgebiete einer Software wie z.B. Frontend oder Datenbanken.
- Microservices haben ein einfaches Routing. Sie haben keine komplizierten, intelligenten Kommunikationsstrukturen mit vielen Regeln zwischen den Prozessen. Hier wird ein alternativer Ansatz verwendet: Intelligente Endpunkte und dumme Verbindungen. Die Anwendungen agieren als Filter, d.h. sie empfangen eine Anfrage, bearbeiten diese und geben eine Antwort zurück.
- Microservices sind dezentralisiert. Wird die Anwendung in verschiedene Komponenten aufgeteilt, muss keine einzelne technische Plattform festgelegt werden, sondern es besteht die Möglichkeit für jeden einzelnen Dienst eine unterschiedliche Technologie zu verwenden. Außerdem sind die Daten dezentralisiert, das heißt jede Zielgruppe kann ihr eigenes Datenmodell haben. So wird gewährleistet, dass die Begriffe auf die Zielgruppe angepasst sind und nicht missverstanden werden. Auch die Datenspeicherung ist dezentralisiert. Es wird keine Datenbank für das gesamte System verwendet, sondern jeder Microservice in einem Softwaresystem verwaltet seine eigene individuelle Datenbank.
- Microservices sollen ausfallsicher sein. Wenn Services als Komponenten benutzt werden, muss das Programm so entworfen sein, dass Ausfälle von Services toleriert werden. Wenn ein einzelner Service ausfällt, sollen benachbarte Dienste trotzdem wie gewohnt weiterarbeiten. Um Ausfälle zu verhindern, gibt es Monitoring-Dienste, deren Aufgabe es ist, den Service zu beobachten und das Risiko eines Ausfalls zu minimieren.
- Microservices helfen dabei, Änderungen einer Software zu kontrollieren, ohne den Entwicklungsprozess negativ zu beeinflussen. Dadurch, dass eine Anwendung in viele Komponenten aufgeteilt wird, bei der jede Komponente austauschbar und erweiterungsfähig ist, ist die Microservice-orientierte Architektur ideal als „Evolutionäres Design“. Evolutionäres Design ist ein Entwurfsansatz, bei dem ein Softwaresystem in kleinen Schritten entwickelt wird und nur absolut notwendige Elemente eingebaut werden. Dadurch ist es leichter, das System in der Zukunft an neue Anforderungen anzupassen. [Evo18]

2.1.4 Webservices

Über die Zeit hinweg hat sich das Internet von einer Anwendung zur Veröffentlichung und zum Finden von Dokumenten auf Webseiten in ein großes programmierbares Medium zum Datenaustausch und zum Abrufen von rechnerfernen Softwarekomponenten, welche als Dienste angeboten werden, verwandelt. Webservices sind Dienste, die die Möglichkeit bieten, unterschiedliche Systeme zu integrieren und wiederverwendbare Geschäftsfunktionen über HTTP bereitzustellen. Dazu wurde standardgemäß der TCP/IP Port 80, welcher zur Benutzung des HTTP Protokolls benötigt wird, von den meisten Firewalls offen gelassen. [FGM+99] HTTP wird hier entweder als einfacher, elektronischer Datentransport z.B. über SOAP/WSDL Dienste benutzt, oder es wird als ein komplettes Anwendungsprotokoll verwendet, in dem die gesamte Semantik des Serviceverhaltens definiert wird. Ein sehr häufig benutztes Beispiel für solch ein Anwendungsprotokoll sind RESTful Webservices. Webservices machen es einfach, gemeinsame Logiken mit unterschiedlichen Clients wie Smartphones, Desktop PCs und auch Webanwendungen zu verwenden und zu teilen. Alle Webservices benutzen HTTP und Standards für den Datenaustausch wie XML und JSON. Sie ermöglichen außerdem relativ einfach die Unterstützung komplexer Geschäftsprozesse durch die Zusammenstellung von Diensten. [Dai11]

Simple Object Access Protocol (SOAP)

SOAP basiert auf dem XML-Nachrichtenformat und ist für die Kommunikation mit Webservices und deren Einbettung in ein Transportprotokoll zuständig. Die SOAP Spezifikation ist in mehrere Dokumente aufgeteilt:

- **Primer:** Ein einfach zu verstehendes Tutorial mit vielen Beispielen, damit die nachfolgenden Spezifikationen besser nachvollzogen werden können.
- **Messaging Framework:** Legt den Rahmen einer SOAP Nachricht fest. Hier werden vorgeschriebene und optionale Elemente der Nachricht beschrieben.
- **Adjuncts:** Hier wird das Datenmodell für SOAP und das Codierungsschema für Methodenaufrufe definiert. Außerdem wird für das HTTP-Protokoll beschrieben, wie die Anbindung an das Transportprotokoll funktioniert.
- Außerdem gibt es die Spezifikationen XML-binary Optimized Packaging, SOAP Message Transmission Optimization Mechanism und Resource Representation SOAP Header Block. Diese sind für Daten zuständig, die nicht in eine XML-Datei geschrieben werden können, wie z.B. Anhänge in Form eines Bildes.

Das Messaging Framework ist für den Aufbau einer SOAP Nachricht zuständig. Prinzipiell ist eine Nachricht ein XML-Dokument bestehend aus drei Teilen: Envelope, Header und Body.

Im Envelope steht die eigentliche Nachricht, es bildet das Wurzelement des Dokuments, d.h. sowohl Header als auch Body sind hierin gekapselt.

Der Header ist ein optionales Element, welches immer das erste Kindelement des Envelopes ist. Hier können zum Beispiel sicherheitsrelevante Informationen beschrieben sein oder der Empfänger bzw. die Zwischenstationen, die dieses Element verarbeiten dürfen, werden genannt.

Der Body ist ein notwendiges Element. Hier stehen die Nutzdaten der Nachricht, also der eigentliche

verwendbare Inhalt. Dieser muss dabei im gültigen XML-Schema sein, um das Marshalling und Demarshalling des Nachrichteninhalts durch die SOAP-Engines der beiden Endpunkte zu ermöglichen. Dadurch kann der Inhalt an die entsprechende Implementierung weitergeleitet werden. [PZL08]

Für den Datenaustausch zwischen zwei Anwendungen können nun unter anderem folgende Protokolle benutzt werden:

- Hypertext Transfer Protocol (HTTP)
- Simple Mail Transfer Protocol (SMTP)
- File Transfer Protocol (FTP)

SOAP-Spezifikationen schreiben nicht vor, welches Protokoll benutzt werden muss, eine SOAP-Nachricht muss lediglich mithilfe eines beliebigen Transportprotokolls übertragen werden. [Mel10]

Web Services Description Language (WSDL)

SOAP bietet zwar die grundlegende Kommunikation an, aber keine Informationen, welche Nachrichten ausgetauscht werden müssen, um erfolgreich mit dem Webservice zu interagieren. Hierfür ist die Web Service Description Language zuständig, die die Schnittstelle des Webservices beschreibt und dem Nutzer einen Ansprechpartner bietet. Ein komplettes WSDL Dokument beinhaltet zwei Arten von Informationsbeschreibungen: eine abstrakte Beschreibung auf funktionaler Ebene und eine konkrete Beschreibung auf technischer Ebene. Durch diese Trennung der zwei Ebenen kann eine wiederverwendbare, modulare Schnittstellenbeschreibung ermöglicht werden.

In der abstrakten Beschreibung gibt es einen sogenannten „portType“, in neueren Versionen auch „interface“ genannt. Er dient zur Beschreibung der generellen Funktionalität eines Webservices und beinhaltet die Menge an Operationen, die unterstützt werden. Diese Operationen definieren sich durch eine Menge an XML-Nachrichten, welche zwischen Anbieter und Nutzer ausgetauscht werden. Außerdem gibt es einen „types“-Abschnitt, in dem die verschiedenen Datentypen definiert werden.

In der konkreten Beschreibung wird im Abschnitt „binding“ das für den Nachrichtenaustausch verwendete Protokoll festgelegt. Außerdem können detaillierte Informationen in Bezug auf Transport und Kodierung beschrieben werden. Zum Schluss muss spezifiziert werden, wo der Webservice physikalisch lokalisiert ist. Mithilfe von „service“-Elementen werden die Endpunkte festgelegt, über die der Dienst erreicht werden kann. [CDK+02] [PZL08]

Weiterhin gibt es noch einen optionalen Abschnitt „documentation“, in dem der Dienst genauer beschrieben werden kann, unter anderem auch, wie der Dienst benutzt werden kann oder wer der Ansprechpartner ist. [Mel10]

2.2 REpresentational State Transfer (REST)

2.2.1 Grundlagen

Das REST Paradigma wurde im Jahre 2000 von Roy Fielding im Rahmen seiner Dissertation vorgestellt. [FT00] REST ist im Gegensatz zu SOAP kein Protokoll, sondern ein Architekturstil. Dieser definiert eine Reihe von Architekturprinzipien, mit denen Webservices entworfen werden können, die sich darauf konzentrieren, wie Ressourcenzustände angesprochen und über HTTP übertragen werden können. Im Gegensatz zu manchen anderen Arten von Webservices unterstreichen REST Webservices die komplette und korrekte Nutzung des HTTP-Protokolls bei der Veröffentlichung von Software im Internet. SOAP zum Beispiel arbeitet hingegen fast ausschließlich mit der POST-Methode. [RR08]

Der Fokus von REST liegt auf Anwendungen im Web, wohingegen SOAP versucht, möglichst viele Arten von verteilten Anwendungen zu unterstützen. [Rod08] [Mel10]

Die leichtgewichtige Infrastruktur von REST hat eine sehr niedrige Adoptionsbarriere, da die Services mit einer minimalen Anzahl an Werkzeugen gebaut werden können und dadurch auch günstig zu beschaffen sind. Der Aufwand für einen Entwickler, einen Client für einen RESTful Webservice zu bauen ist gering, da HTTP-Anfragen mithilfe von Werkzeugen wie Postman einfach zu verschicken sind und dadurch der Webservice einfach zu testen ist. Außerdem können GET-Anfragen auf einem normalen Webbrowser getestet werden. [PZL08]

Über die Zeit wurde REST immer beliebter und ersetzt mehr und mehr die damals oft benutzten Webservices mit SOAP. Eine Google Trendanalyse, die analysiert hat, wie oft nach den Schlüsselwörtern REST und SOAP gesucht wurde, unterstützt diese These. Außerdem geben immer mehr Webseiten im Internet an, dass sie nun auf REST aufbauen. Sogar große Firmen wie Amazon und Twitter benutzen REST-ähnliche Schnittstellen. [Mas11]

2.2.2 Design Prinzipien und Richtlinien

Eine konkrete Implementierung eines REST Webservice kann verschiedenen Design Prinzipien folgen. Diese engen Vorgaben führen dazu, dass die Dienste gut strukturiert sind. Im Folgenden werden wichtige Design Prinzipien von [Rod08], [PZL08] und [Pau14] gelistet:

- Das Benutzen einer einheitlichen Schnittstelle mit einer ausreichenden Anzahl an Methoden, die sämtliche mögliche Interaktionen mit Ressourcen abdecken. Bei RESTful Webservices wird das HTTP Protokoll benutzt, welches unter anderem die Operationen GET, PUT, POST, DELETE umfasst. POST erstellt dabei eine neue Ressource, welche von DELETE gelöscht werden kann. GET ruft den aktuellen Zustand der Ressource ab und PUT kann darauf einen neuen Zustand übertragen. Wenn nötig kann das Protokoll durch weitere Methoden erweitert werden.
- Jede Interaktion mit einer Ressource ist zustandslos, was bedeutet, dass in jeder Anfrage sämtliche Daten, die benötigt werden, mitgeschickt werden. Dadurch wird keine dauerhafte Verbindung zwischen Server und Client benötigt und der Server muss beim Bearbeiten der Anfrage keinen Anwendungskontext oder -zustand abrufen. Diese Zustandslosigkeit ermöglicht

ein horizontales Skalieren des Systems und vereinfacht das Design und die Implementierung von serverseitigen Komponenten, da der Server keine Daten mit einer externen Applikation synchronisieren muss.

- Die Identifikation von Ressourcen erfolgt durch eindeutige verzeichnisstruktur-ähnliche Uniform Resource Identifiers (URI). Dabei sollen diese URIs intuitiv, unkompliziert und leicht verständlich sein. Es sollen keine Erklärungen oder Referenzen nötig sein, um zu erkennen, wohin die URI zeigt.
- Die Nachrichten sind selbstbeschreibend. Eine Repräsentation der Ressource spiegelt typischerweise den aktuellen Zustand einer Ressource und ihrer Attribute in dem Moment, in dem eine Anwendung sie anfordert, wider. Sie sind somit nur Momentaufnahmen. Auf ihre Inhalte kann durch Anpassen des Werts des „HTTP Accept Header“ in verschiedenen gewünschten Formaten zugegriffen werden, wie z.B. HTML, XML oder JSON. Dies ermöglicht die Nutzung des Services von einer Vielfalt an Clients aus, die in verschiedenen Sprachen geschrieben sind und auf verschiedenen Plattformen und Geräten laufen. Dadurch, dass der Client das Datenformat aussuchen kann, wird auch die Datenkopplung zwischen dem Dienst und der Anwendung reduziert.

Wichtig ist, sich nicht ausschließlich auf Richtlinien zu verlassen. Nach Mulloy [Mul12] ist die Nutzbarkeit des Webinterfaces wichtiger als solche Richtlinien, denen blind gefolgt wird. Dies führt jedoch wiederum zu dem Problem, dass einige Konzepte von REST nicht mehr eingehalten werden, was die Folge hat, dass die Schnittstelle nach anderen Richtlinien nicht mehr als REST-konform angesehen wird.

Richardson et al. [RAR13] schlugen eine andere Herangehensweise an das Thema vor. Sie haben sich tiefergehend mit dem Verständnis von REST selbst beschäftigt, anstatt sich auf Richtlinien und Prinzipien zu konzentrieren. Außerdem haben sie zum ersten Mal den Schwerpunkt auf Hypermedia und das dazugehörige „Hypermedia As The Engine Of Application State“ (HATEOAS) gesetzt, was zuvor in den meisten Richtlinien vernachlässigt wurde.

Das war auch für Roy Fielding, dem Erschaffer der REST-Architektur, der wichtigste Aspekt von REST. Laut ihm ist eine REST-Schnittstelle, die nicht auf dem Prinzip des HATEOAS basiert, keine richtige REST-Schnittstelle. Auch im Richardson-Reifegradmodell ist dieser Aspekt von sehr großer Bedeutung, wie im nächsten Abschnitt gezeigt wird. [Roy08]

Um HATEOAS zu verstehen, muss die Bedeutung von Hypermedia bekannt sein. Hypermedia bezieht sich auf Inhalte, die in ihrer Ressourcenrepräsentation oder in den dazugehörigen Metadaten Pfade zu verwandten Ressourcen, die als nächstes aufgerufen werden können, enthalten. HATEOAS basiert darauf, dass der Client URI Pfade im Hypermedia-Format als Antwort von Anfragen bekommt, sodass er dynamisch durch Klicken auf diese Pfade zu seiner gewollten Ressource hin navigieren kann. Wenn beispielsweise über eine POST-Anfrage ein lang andauernder Prozess initialisiert wird, werden bei der Antwort verschiedene Pfade zurückgegeben. Zum Beispiel kann über einen der zurückgegebenen Pfade der Status des Prozesses abgefragt werden und mit einem weiteren Pfad der Prozess gestoppt werden. Pfade können außerdem auch zu Multimedia Ressourcen wie z.B. Bildern oder Videos führen. [Restapi18]

Der Vorteil hierbei ist, dass die Ressource die komplette Kontrolle über den Zugriff auf sich besitzt, da sie lediglich eine Repräsentation von sich weitergibt. Die Repräsentation kann vom Client beliebig manipuliert werden, ohne dass es negative Auswirkungen auf die Ressource selbst hat. Das führt dazu, dass die Kopplung zum Client minimal ist, während die Autonomie trotzdem maximal bleibt.

Im Gegensatz dazu kommunizieren Webservices, welche auf SOAP basieren, über ein festes Interface. Jedes Mal, wenn ein Service geändert wird, muss dabei eine neue Schnittstelle mit einer neuen Beschreibung bereitgestellt werden.

Ein weiterer Vorteil ist, dass dank den Hyperlinks Ressourcen gefunden werden können, ohne dass eine zwangshafte Registrierung zu einer zentralisierten Repository benötigt wird. [PZL08]

Obwohl die Bedingung, dass RESTful Webservices HATEOAS benutzen sollen, sehr nützlich ist, ist es trotzdem die am wenigsten eingehaltene Bedingung. [Jax14] [Pau14]

2.2.3 Richardson-Reifegradmodell

Richardson hat ein Modell entwickelt, mit dem eingestuft werden kann, wie sehr die Bedingungen von REST von der API eingehalten werden. Je besser die API diese Bedingungen erfüllt, desto höher ist auch das Level des Reifegradmodells. Das Modell geht von Level 0 bis Level 3.

Dies führt zu der Frage, ob nur Dienste, die auf dem höchsten Level sind, RESTful genannt werden dürfen. Zurzeit behaupten jedoch viele APIs mit einem niedrigen Level, dass sie Rest-konform sind. Die Level werden in [Fow10], [WPR10] und [Pau14] genauer beschrieben:

Level 0 - Sumpf des POX (Plain Old XML)

Level 0 setzt das Benutzen von HTTP als Transportsystem für Interaktionen voraus, jedoch ohne die Mechanismen des Internets zu benutzen. HTTP fungiert hier lediglich als Tunnel für Anfragen und Antworten durch einen HTTP Endpunkt, ohne dass das Protokoll den derzeitigen Anwendungszustand anzeigt. Hier werden XML Dokumente über eine einzelne HTTP Methode, meistens über die POST-Methode, ausgetauscht. Dabei werden die anderen HTTP Methoden ignoriert, was zur Folge hat, dass nicht das volle Potential des HTTP Protokolls ausgeschöpft wird. Beispielsweise wird XML-RPC oder POX verwendet, die POST-Anfragen mit XML Payloads durch einen einzelnen URI Endpunkt schicken. Die Antworten sind dann auch wieder in XML. Das XML Payload kann auch durch JSON, YAML oder ähnliche leichtgewichtige Formate ersetzt werden, wodurch die Performanz des Services optimiert werden kann. Da alle Nachrichten durch den gleichen Endpunkt gehen, kann der Service nur durch Parsen der Datei zwischen verschiedenen Operationen unterscheiden.

Level 1 - Ressourcen

Auf diesem Level wird weiterhin nur eine einzelne HTTP Methode benutzt, aber die API ist in der Lage, Ressourcen zu verwenden. Über mehrere URIs können nun eindeutig Ressourcen zugeordnet werden. Im Gegensatz zu Level 0, wo es nur möglich war, Daten zwischen den Service-Endpunkten auszutauschen, ist es nun möglich, Anfragen an viele gezielte Ressourcen zu schicken. Dadurch wurde eine Art Objektidentität geschaffen, die es ermöglicht, Methoden für ein bestimmtes Objekt und Argumente an eine URI weiterzugeben, die für das Objekt zuständig sind. Daraufhin kann die URI an einen entfernten Dienst weitergeschickt werden, meistens über die HTTP GET Methode.

Level 2 - HTTP Verben

Services auf diesem Level hosten viele durch URI adressierbare Ressourcen. Dabei werden von jeder Ressource alle HTTP Verben akzeptiert. Das bedeutet nicht nur, dass Clients die HTTP Methoden wie zum Beispiel GET, PUT, POST und DELETE benutzen können, sondern auch, dass die Methoden semantisch korrekt sein sollten. Das hat zur Folge, dass die zu den Methoden gehörende Sicherheit und Idempotenz benutzt werden kann, um das System entsprechend zu optimieren. Beispielsweise ist die GET-Methode definiert als sichere Methode ohne Nebeneffekte, welche außerdem gecached werden kann. Dies führt dazu, dass der Verarbeitungsaufwand der API deutlich reduziert werden kann. Zusätzlich wird auch die Transiteffizienz gesteigert, da es nicht mehr notwendig ist, auf Anfragen, welche die gleichen Antworten generieren, zu antworten. [Nordic18] Ein weiterer Vorteil ist, dass fehlgeschlagene DELETE- und PUT-Methoden automatisch wiederholt werden können. Desweiteren können Services auf Anfragen entsprechende HTTP-Statuscodes zurückgeben, sodass zum Beispiel erkannt werden kann, welcher Service für eine gescheiterte Interaktion zuständig war.

Level 3 - Hypermedia Bedienungselemente

Das höchste Level implementiert Hypermedia und das dazugehörige HATEOAS wie im vorherigen Kapitel beschrieben. Dies öffnet der API viele neue Funktionalitäten. Durch HATEOAS können die API Antworten mit zusätzlichen Informationen zurückgegeben und verschiedene Ressourcen miteinander verbunden werden, wodurch eine bessere Interaktion und damit auch ein reicheres Benutzererlebnis ermöglicht wird. [Nordic18] Anstatt von Vornherein zu wissen, welche Ressourcenadressen benutzt werden, kann ein Client nun dynamisch entscheiden, mit welchen Ressourcen er interagieren will, indem er eigenständig Pfade durchgeht. Zusätzlich können auch die URI Schemas geupdated werden, ohne dass es Probleme mit dem Client gibt, sodass es größere Freiheiten bei der Backendentwicklung gibt. [Nordic18]

Auswirkungen der Reifegradlevel

Das Reifegradlevel eines Services ist nicht nur als Maßstab zu sehen, wie REST-konform die API ist, sondern beeinflusst auch die generelle Qualität der Architektur.

Zum Beispiel führt das Tunneln von Level 0 durch eine einzige HTTP-Methode dazu, dass lediglich elementare Kommunikation und Datenaustausch möglich sind. Daraus folgt ein brüchig integriertes System, was die Veränderbarkeit, Weiterentwickelbarkeit und Skalierbarkeit in der Zukunft erschwert.

Durch die Unterscheidung der Ressourcen auf Level 1 wird es Services ermöglicht, jede Ressource über ihre URI zu adressieren. Außerdem ist es nun möglich, das Teile-und-Herrsche-Prinzip auf das Design der Serviceschnittstelle anzuwenden. Dieses Prinzip erlaubt es, ein großes, komplexes Problem so lange rekursiv in kleinere Teilprobleme zu zerlegen, bis es möglich ist, es zu lösen. Aus diesen einzelnen Lösungen wird dann eine Lösung für das Gesamtproblem ermittelt. [FBT15] Damit können kleinere Teile der Schnittstelle betrachtet werden, wodurch es einfacher wird, die Schnittstelle zu verstehen.

APIs von Level 2 sind einheitlich und standardisiert. Es soll keine redundanten Variationen von

Ressourcen geben, sondern die Wiederverwendung von Ressourcen soll gefördert werden. Außerdem können alle Services mit allen Ressourcen kommunizieren, was wiederum zugunsten der Interoperabilität geht.

Das Benutzen von Hypermedia auf dem höchsten Level wurde schon von Roy Fielding als wichtigste Voraussetzung des REST Designs erklärt. Es ermöglicht APIs auf eine natürliche und effektive Weise zu antworten. [Nordic18] Außerdem minimalisiert die Verwendung von Hypermedia die Kopplung zwischen Service und Client, was in der Service-orientierten Architektur von großer Bedeutung ist.

Da RESTful Webservices Caching, Cluster und Lastverteilung unterstützen, sind sie gut skalierbar und haben auch keine Probleme bei einer großen Menge an Clients. [PZL08] [Pau14]

2.2.4 Vergleich REST mit WS-*

WS-* sind Spezifikationen, die auf Webservices mit SOAP/WSDL aufbauen. Sie erweitern SOAP/WSDL und beziehen sich auf Anwendungsgebiete, die dort nicht festgelegt wurden. Es ist auch möglich, viele Spezifikationen zu kombinieren.

Pautasso et al. haben in ihrem Paper „RESTful Web Services vs. Big Web Services: Making the Right Architectural Decision“ [PZL08] eine Gegenüberstellung von REST und WS-* vorgenommen. Der Vergleich ist in folgende Abstraktionslevel unterteilt: Architekturprinzipien, konzeptionelle Entscheidungen und technologische Entscheidungen.

Architekturprinzipien

Im Architekturprinzip „Protocol Layering“ wird angeschaut, ob HTTP als eine Anwendung oder ein Transportprotokoll benutzt wird.

Bei REST wird das Web als ein universelles Medium angesehen, um global zugängliche Informationen zu verbreiten. Durch das Benutzen von URIs sind die Anwendungen ein Teil des Webs. Auch werden hier alle HTTP Verben benutzt, um auf die Ressourcen zuzugreifen, sie zu verändern oder zu löschen.

Anders ist es bei SOAP. Hier wird das Web als Transportmedium für Nachrichten angesehen. Das bedeutet, dass die Anwendungen nur von außen mit dem Web interagieren, aber nicht Teil des Webs sind. Dies kann dadurch erkannt werden, dass nur die Endpunkte der Webservices miteinander kommunizieren und auch nur ein HTTP Verb benutzt wird. Somit werden alle Informationen über die Operationen des Services in die SOAP Nachricht geschoben.

Im nächsten Architekturprinzip wird betrachtet, wie REST und SOAP mit Heterogenität umgehen, zum Beispiel mit unterschiedlichen Browsern.

Verschiedene Browser-Anbieter benutzen unter anderem unterschiedliche Bibliotheken oder rendern HTML-Seiten unterschiedlich. Was jedoch alle Browser immer gleich anbieten ist das selbe HTTP Protokoll. Da bei REST das HTTP Protokoll vollkommen ausgeschöpft wird, kann REST sehr gut mit Heterogenität von Browsern umgehen, wohingegen dies bei SOAP nicht der Fall ist, da es nur ein einziges HTTP Verb verwendet.

Womit SOAP und WS-* jedoch gut umgehen können, ist mit der Domäne des Enterprise Computing,

welche aus einer Ansammlung an heterogenen, autonomen und verteilten Systemen besteht. Die Softwaresysteme, die hier zu finden sind, sind oftmals seit den Anfängen des Webs vorhanden und wurden in vielen unterschiedlichen Technologien implementiert.

Das letzte Prinzip, das betrachtet wird, ist das der losen Kopplung. Sowohl WS-* als auch REST fördern die Entwicklung von lose gekoppelten Systemen, jedoch liegt der Fokus auf unterschiedlichen Aspekten. Ein wichtiger Aspekt von loser Kopplung ist der Umgang mit nicht erreichbaren Serviceanbietern. Bei einem lose gekoppelten System sollen die Clients nicht beeinträchtigt werden, wenn der Service kurzzeitig nicht erreichbar ist. Dies ist der Fall bei SOAP, da hier Nachrichten geschickt werden, die bei Nichterreichbarkeit des Ziels in einer zuverlässigen Warteschleife landen. RESTful Webservices hingegen benutzen eine synchrone Interaktion, das heißt, wenn der Serveranbieter nicht erreichbar ist, kann auch keine Interaktion stattfinden. Ein weiteres Merkmal für lose Kopplung ist die Evolvierbarkeit von Webservices, das heißt es ist möglich, diese zu verändern, ohne die Clients zu beeinflussen. Bei REST werden die HTTP Verben benutzt, die standardmäßig definiert sind und sich niemals ändern. Dies erlaubt eine komplette Entkopplung von Server und Client. Bei WS-* ist dies nicht so einfach, da jede Serviceschnittstelle ihre eigene Menge an Operationen enthält, die sich auch verändern kann. Jedoch wird das Format des XML-Payloads, wie unter anderem die Syntax, die Struktur und die Semantik, genau spezifiziert. Damit teilen sich sowohl WS-* als auch REST die Eigenschaft der losen Kopplung von XML. Die Benutzung von SOAP Nachrichten ermöglicht damit eine leichte Modifikation der Serviceschnittstelle, ohne diese zu zerstören.

Konzeptionelle Entscheidungen

Es ist wichtig, dass die Beschreibung des Serviceinterface klar definiert und maschinenlesbar ist. In WS-* sind dafür folgende zwei Vorgehensweisen entstanden: Contract-first und Contract-last. Bei Contract-first wird zuerst die Spezifikation der Schnittstelle in Form eines WSDL Vertrags erstellt und aus dieser Spezifikation das System implementiert. Im Gegensatz dazu wird bei Contract-last zuerst die Software implementiert und der WSDL Vertrag daraus automatisch generiert. REST hingegen beschränkt die Schnittstelle der Ressource immer auf eine generische einheitliche Form mit vordefinierten Methoden. Hier muss sich der Designer nur auf das Definieren und Implementieren der freigelegten Ressourcen konzentrieren.

Wenn nun nach der Komplexität des Schnittstellendesigns geurteilt wird, ist REST einfacher als WS-*, da die Operationen klar definiert und begrenzt werden. Bei WS-* Webservices ist damit das Design der Schnittstelle eine wichtige Entscheidung, da dieses sämtliche Operationen des Services definiert. Da jede Schnittstelle unterschiedliche Operationen haben kann, gibt es auch keine vordefinierte Semantik für die Operationen und es ist umso wichtiger, die Funktionalität des Services in einer nachvollziehbaren Weise zu beschreiben.

Pautasso et al. fanden auch heraus, dass beim Designen eines RESTful Webservice viele Architektur-Entscheidungen getroffen werden müssen, aber für diese Entscheidungen wenige Alternativen zur Auswahl stehen. Sie fanden acht Entscheidungen mit insgesamt zehn Alternativen. Die Auswahl einer Alternative ist außerdem mit einem erheblichen Aufwand verbunden.

Im Gegensatz hierzu gibt es bei WS-* nur fünf Entscheidungen zu treffen, jedoch existieren hier insgesamt mehr Alternativen. Die Alternativen sind einfacher zu implementieren, da die Konzepte

standardisiert sind und hohe Werkzeugunterstützung in WS-* existiert. Demnach gibt es hier eine größere Entscheidungsfreiheit als bei REST, aber mit relativ strengen konzeptionellen Grenzen durch die Spezifikation.

Technologische Entscheidungen

Beim Thema Transportprotokoll gibt es bei RESTful Webservices keine andere Möglichkeit, als HTTP zu benutzen. Daher ist hier keine Entscheidung über die Wahl des Kommunikationsprotokolls nötig. Im Gegensatz dazu können bei WS-* die SOAP Nachrichten mit einer Vielfalt an Transportprotokollen, unter anderem natürlich auch mit dem HTTP Protokoll, verschickt werden. Es ist bei WS-* außerdem möglich, zwischen synchroner und asynchroner Kommunikation zu wählen.

RESTful Webservices haben kein eindeutiges Format um ihre Ressourcen zu repräsentieren. Hier kann zwischen einer Vielfalt an Multipurpose Internet Mail Extensions (MIME) Dokumenttypen gewählt werden. WS-* Webservices haben ein einheitliches Nachrichtenformat mit SOAP.

Bis vor kurzem wurden RESTful APIs ausschließlich anhand von menschenlesbaren und informellen Dokumenten beschrieben. Jedoch kamen in den letzten Jahren mit der Swagger/OpenAPI Spezifikation und RAML zwei erfolgreiche maschinenlesbare, einheitliche Beschreibungsformate für REST APIs auf den Markt. WS-* hat mit WSDL Dokumenten ebenso ein einheitliches maschinenlesbares Format.

Interessant ist auch, dass SOAP Webservices verglichen mit RESTful Webservices einen deutlich höheren Netzwerkverkehr, höhere Wartezeiten und einen deutlich größeren Nachrichtenumfang haben. Dagegen ist REST leichtgewichtiger, einfacher und hat eine höhere Flexibilität und weniger Verwaltungsdaten. [MP+13]

Bei REST wird der URI Standard benutzt, um Ressourcen zu adressieren. Bei WS-* dagegen gab es keinen eindeutigen Mechanismus und es wurde auch auf URI zurückgegriffen. Seit neuerem gibt es das sogenannte *WS-Addressing*, eine modulare Erweiterung für SOAP. Hier wird ein Standard für die Integration von Nachrichtenadressierungsinformationen in den Nachrichten definiert. Damit wird eine einheitliche Adressiermethode für SOAP Nachrichten angeboten.

2.3 Swagger/OpenAPI

Swagger wurde im Jahre 2010 als einfache Open-Source Spezifikation zur Beschreibung von RESTful APIs gegründet. [Swa18]

Die Swagger Spezifikation ist die Parallele zu den WSDL Dokumenten von SOAP-basierten Webservices. In dem Swagger-Dokument werden die angebotenen Ressourcen der RESTful API und die darauf anwendbaren Operationen spezifiziert. Außerdem werden auch Informationen über die benötigten Parameter der Operationen definiert, wie zum Beispiel den Namen und den Typ des Parameters, ob der Parameter optional ist und welche Werte akzeptiert werden. Zusätzlich können die Strukturen des Request- und des Response-Bodys in einem JSON Schema beschrieben werden. [Sur16]

Im Jahr 2015 wurde das Swagger Projekt von dem Unternehmen SmartBear Software übernommen. Die Spezifikation wurde an die Linux Foundation übergeben und in „OpenAPI Spezifikation“ (OAS) umbenannt. Darauf folgend wurde die „OpenAPI Initiative“ gegründet, die sich die Unterstützung

und Entwicklung der OpenAPI Spezifikation zur Aufgabe gemacht hat. [Swa18] Diese OpenAPI Initiative wird von über 30 großen Organisationen geleitet, darunter sind auch Industrieführer wie Google, Microsoft und IBM. SmartBear selbst investiert auch weiter in die Swagger Community, das Swagger Ökosystem und die Werkzeuge, die die Spezifikation unterstützen. [SwaPre18]

Um die in der Spezifikation definierten Aspekte besser implementieren zu können, wurden zusätzlich Werkzeuge von den Erzeugern der ehemaligen Swagger Spezifikation bereitgestellt. Diese Werkzeuge können in unterschiedlichen Stadien des API Lebenszyklus eingesetzt werden und haben verschiedene Funktionen:

- **Swagger Editor** zum Erstellen und zur Bearbeitung von OpenAPI Spezifikationen im Browser. Die Spezifikation wird dabei in Echtzeit visualisiert, sodass der Entwickler direkt mit der API interagieren kann.
- **Swagger UI** zur automatischen Generierung einer visuellen Dokumentation aus einer OpenAPI Spezifikation.
- **Swagger Codegen** zur Generierung von API Client Bibliotheken, Server Stubs und Dokumentationen.
- **Swagger Parser** zum Parsen des OpenAPI Dokuments in Java.
- **Swagger Core** zum Erstellen, Nutzen und Arbeiten mit OpenAPI Definitionen über Java-basierte Bibliotheken.
- **Swagger Inspector** zur Generierung von OpenAPI Definitionen aus einer OAS-konformen API.
- **SwaggerHub** zum Arbeiten am API Design und zur automatischen Generierung einer API Dokumentation aus dem erstellten Design. Mithilfe des SwaggerHubs können außerdem verschiedene Teams einfacher zusammenarbeiten.

Zusätzlich zu den Werkzeugen der Swagger-Erschaffer gibt es eine große Menge an Third-Party Entwicklern, die ihre eigenen Lösungen für die Unterstützung der OpenAPI Spezifikation bereitstellen. [SwaOAS18]

2.3.1 Code First und Design First

Es gibt zwei verschiedene Arten, APIs mithilfe der OpenAPI Spezifikation zu entwickeln: Code First und Design First.

Code First ist die traditionelle Herangehensweise, bei der zuerst die Geschäftserfordernisse definiert werden und daraufhin der Code geschrieben wird. Aus dem Code wird danach eine Dokumentation, hier in Form des Swagger Dokuments, mithilfe von Werkzeugen wie Swagger Inspector generiert.

Design First ist eine neuere Herangehensweise. Hier wird zuerst der Design-Plan in einem API Vertrag definiert und daraufhin der Code mithilfe des Vertrags geschrieben. Der Vertrag ist dabei sowohl menschen- als auch maschinenlesbar. Der Entwickler wird beim Coden durch Werkzeuge wie Swagger Codegen, der automatisch Server Stubs und Client SDKs generieren kann, unterstützt. [DFoCF18]

Beide Herangehensweisen haben ihre Vor- und Nachteile. Die Nutzer der API und ihre Bedürfnisse spielen hier eine große Rolle. Es ist auch wichtig zu überlegen, welche Probleme mit der API gelöst werden sollen. Im Folgenden werden Gründe für die Wahl der beiden Methoden aufgezählt.

Gründe für Code First

Der Code First Ansatz wird gewählt, wenn die Marktstrategie Fokus auf Geschwindigkeit und Agilität legt. Entwickler können die API viel schneller implementieren, wenn sie direkt aus den Geschäftserfordernissen programmiert wird. Da Code First die traditionelle Herangehensweise ist, gibt es hier auch viele Bibliotheken, die die Implementierung unterstützen, wie zum Beispiel Bibliotheken für funktionales Testen oder zum automatischen Deployen von Daten. [DFoCF18]

Eine weitere Situation, in der der Code First Ansatz vorteilhaft ist, ist bei einer einfachen REST API. Wenn die API sehr einfach ist, ist die Erstellung eines Designs überflüssig und der Designprozess kann übersprungen werden, sodass direkt mit dem Coden gestartet werden kann. Es ist auch sinnvoll direkt zu coden, wenn auf einem anderen Projekt aufgebaut wird und dieses nur leicht modifiziert werden muss. [NDFCF18]

Ideal ist der Code First Ansatz auch, wenn die API nur von dem Team oder der Firma benutzt wird, die diese API gebaut haben. Hier liegt der Fokus eher auf der Funktionalität als auf dem Design, da die API nicht für Kunden, also für Clients, die sich nicht mit der Nutzung des Dienstes auskennen, gedacht ist. Das ist insbesondere der Fall bei einer kleinen API mit wenigen Endpunkten, die nur intern benutzt wird. [DFoCF18]

Wenn das API Framework keine automatische Code Generierung unterstützt, macht es Sinn, den Code First Ansatz zu wählen. Das Design Dokument, welches beim Design First Ansatz erstellt wird, wäre hier lediglich für die Dokumentation und Kommunikation zuständig und kann nicht zur Code Generierung verwendet werden. Der Code muss daher trotzdem manuell implementiert werden. Die Anforderungen an den Code können bei nicht vorhandenem Design Dokument direkt aus den Geschäftserfordernissen gelesen werden. [NDFCF18]

Gründe für Design First

Beim Design First Ansatz liegt der Fokus auf dem Design der API. Der API Vertrag, der bei diesem Ansatz erstellt wird, führt zu einer besseren Developer Experience, was wiederum die Arbeit der Entwickler erleichtert und damit die Motivation fördert. Ein effektives API Design reduziert die Lernkurve und damit auch die Zeit, die der Nutzer für die Integration der API benötigt. Das ist vor allem dann wichtig, wenn die Zielgruppe der API externe Kunden sind. In diesem Fall ist die API die Hauptmethode, wie die Kunden mit den angebotenen Services kommunizieren. Ein gutes Design fördert die Kundenbindung und spielt eine wichtige Rolle bei der Repräsentation der Organisation.

Der menschenlesbare API Vertrag kann leichter auf Fehler und mögliche Probleme durchsucht werden, da dadurch alle beteiligten Personen des Projektes in der Lage sind, den Vertrag auf die Ziele der API und die Ressourcen hin zu überprüfen. Bei Code First werden die Bugs und die Design-

probleme oft erst nach Beendigung der Implementierung entdeckt. Durch den API Vertrag können Teams in einem früheren Stadium miteinander interagieren, wodurch das Qualitätssicherungsteam frühzeitig eingeschaltet werden kann. [DFoCF18]

Manche Organisationen benutzen die verbraucherorientierte Vertragsgestaltung, bei der eng mit den Verbrauchern und ihren Anforderungen an die API zusammengearbeitet wird. Die API versucht dann alle Anforderungen zu erfüllen. Hier ist der menschenlesbare API Vertrag das wichtigste Medium zur Kommunikation mit den Kunden.

Wenn das Framework einen Codegenerator zur Verfügung stellt, ist es sinnvoll, diesen auch zu benutzen. Das ist effizienter als den Code manuell zu schreiben und verringert die Wahrscheinlichkeit, Bugs einzubauen. Außerdem ist es auch nützlich für die Wartbarkeit. Soll die API in der Zukunft verändert werden, muss sich nicht in den Programmcode eingearbeitet werden, sondern es existiert ein menschenlesbarer Vertrag, der angepasst werden kann. Mithilfe dieses Vertrags kann der Code mit den Anpassungen neu generiert werden.

Das Verwenden des Design First Ansatzes ist auch sinnvoll, wenn ein traditionelles Softwaresystem in ein System umgewandelt werden soll, welches mit Microservices arbeitet. Es macht hier Sinn, schon vorher alle Services in einem Vertrag zu definieren, sodass diese unabhängig voneinander programmiert werden können. [NDFCF18]

2.3.2 Aufbau der Spezifikation

Die aktuellste Version der OpenAPI Spezifikation ist Version 3.0.1. Das OpenAPI Dokument ist ein JSON Objekt, welches entweder im JSON oder YAML Format repräsentiert wird. In der folgenden Abbildung ist die Struktur der Spezifikation der Versionen 2 und 3 abgebildet.

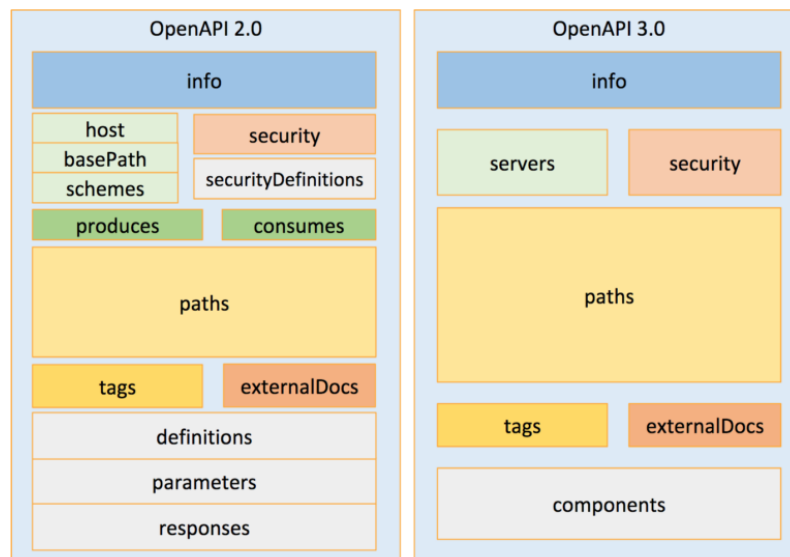


Abbildung 2.1: Struktur der OpenAPI Spezifikationen Version 2 und 3 [OAI17]

Wie in der Abbildung 2.1 erkennbar ist, wurde die Spezifikation klarer strukturiert. Änderungen sind unter anderem folgende [EntOAS18]:

- *Definitions, parameters* und *responses* wurden zusammengefasst zu *components*.
- In der OpenAPI Spezifikation Version 2 kann nur ein *host* definiert werden, wohingegen in der neuen Version zusätzliche Zielsever angegeben werden können. *Hosts, basePath* und *schemes* wurden zusammengefasst zu *servers*.
- In Version 2 können Beschreibungen nur auf Operationsebene angegeben werden. Dies hat dazu geführt, dass unterschiedliche Operationen auf demselben Pfad Duplikate erzeugt haben. In OpenAPI Version 3 kann nun pro Pfad eine ausführliche Beschreibung angegeben werden.
- In Version 3 können Beispiele beschrieben werden. Diese werden entweder als String oder als externe Referenz angegeben.
- In Version 2 war der Mediatyp auf einen einzigen Typ beschränkt. Seit der dritten Version kann pro Pfad ein Mediatyp im *content*-Objekt definiert werden.
- In Version 2 konnten nur einfache Request/Response-HTTP definiert werden. Ab Version 3 werden nun auch weitere Muster unterstützt.

Struktur der OpenAPI Spezifikation 3.0

Das Wurzelement des OpenAPI Dokuments ist das OpenAPI Objekt. Dieses beinhaltet sämtliche Objekte von OpenAPI 3.0, die in der Abbildung 2.1 dargestellt sind. Das OpenAPI Objekt beginnt mit der **Versionsnummer**. Diese Version definiert die Gesamtstruktur der Spezifikation. Seit Version 3 besteht die Version aus drei Komponenten.

Der **info**-Bereich gibt Entwicklern eine high-level Übersicht über die API und enthält Metadaten der API. Hier muss ein Titel, eine Version und eine Beschreibung der API vorhanden sein. Weitere optionale Komponenten des Bereichs sind Kontaktinformationen, Lizenzinformationen und Nutzungsbedingungen.

Das **servers**-Objekt spezifiziert den API Server und gibt dem Client Informationen über den Standort des Servers in Form einer Uniform Resource Locator (URL). Alle später in der Spezifikation vorkommenden API Pfade sind relativ zu der Server URL. Dabei können auch mehrere Server URLs definiert werden. Das ist wichtig, weil APIs in mehreren Umgebungen gleichzeitig existieren können und sich die Geschäftslogik auch an die Umgebung anpassen kann.

Im **security**-Bereich werden die Authentifizierungs- und Autorisierungsmethoden, die in der API benutzt werden, beschrieben. OpenAPI unterstützt dabei unterschiedliche Verfahren, sodass keine ungewollten Nutzer auf die API zugreifen können. Beispiele für solche Verfahren sind: HTTP Authentifizierung, OAuth2 oder OpenID Connect Discovery. Das *security*-Objekt ist jedoch nur zum Aufruf der aktuellen Sicherheitsdefinitionen zuständig, die eigentlichen Sicherheitsimplementationen werden im *components*-Bereich erstellt.

Das **paths**-Objekt zeigt die offenliegenden Endpunkte der API und die dazu gehörenden HTTP-Methoden. Unter jeder Methode stehen außerdem die Anfragen und Antworten der Methode. Für die Anfrage kann das *requestBody*-Schlüsselwort benutzt werden. Hier wird der Inhalt der Anfrage

und der Medientyp beschrieben. Anfragen können außerdem Parameter enthalten, die im *Parameter*-Objekt definiert werden. Die Parameter können dabei über vier verschiedene Methoden übergeben werden: *URL path*, *query string*, *headers* oder *cookies*. Das *Parameter*-Objekt kann auch noch weitere Informationen beinhalten, wie zum Beispiel den Datentyp, das Format oder ob es optional ist oder zwingend benötigt wird. Die Antworten auf diese Anfragen werden im *response*-Objekt beschrieben. Für jede Operation gibt es hier mögliche HTTP-Statuscodes wie zum Beispiel *200 OK* oder *404 Not Found*, die zeigen, ob die Anfrage erfolgreich war oder nicht. Für jeden Statuscode erscheint außerdem eine Antwort in Form eines *response body*-Schema, in dem Informationen über die Antwort des vorliegenden Status enthalten sind.

Im **externalDocs**-Abschnitt werden alle zusätzlichen Informationen, die die Integration mit der API erleichtern, beschrieben. Hier können auch Referenzen zu anderen externen Dokumentationen enthalten sein.

Tags werden benutzt, um Operationen logisch zu gruppieren. Für die Gruppierung kann eine Kategorie definiert und Operationen den Kategorien zugewiesen werden. Dafür kann in den *path*-Operationen eine Liste an Tags definiert werden, an die die *tags*-Objekte automatisch angehängt werden. In diesen *tags*-Objekten können dann weitere Informationen über das Tag beschrieben werden.

Der **components**-Bereich enthält eine Menge an wiederverwendbaren Komponenten des API Designs, wie zum Beispiel Schemas, Antworten, Parameter oder Beispiele. Dadurch kann die Spezifikation gekürzt werden und es gibt keine Wiederholungen von existierenden Elementen. In jedem *path*-Element können Komponenten über das Schlüsselwort `$ref` referenziert werden. [OAbs18] [OAdap18]

2.4 Wartbarkeit

Ein Softwarelebenszyklus kann in zwei große Phasen aufgeteilt werden: die Phase der Softwareentwicklung und die der Softwarewartung. Die Softwarewartung bezieht sich einerseits auf die Aktivitäten, die durchgeführt werden, um ein System einsatzfähig zu halten, andererseits umfasst sie auch die Weiterentwicklung des Systems. Die Phase der Wartung beginnt in dem Moment, in dem das System zum ersten Mal veröffentlicht wird. [TS12]

Lientz et al. [LS80] haben die Wartung in vier große Kategorien aufgeteilt.

- **Corrective maintenance:** Diese Art der Wartung wird durchgeführt, nachdem das Projekt ausgeliefert ist und ein Problem entdeckt wird. Es werden Bugs im Code korrigiert.
- **Adaptive maintenance:** Bei einzelnen oder ständigen Änderungen der Umgebung nach der Auslieferung der Software wird Adaptive maintenance betrieben. Die Veränderungen bei dieser Art der Wartung erlauben dem System in einer neuen technischen Infrastruktur zu laufen.
- **Perfective maintenance:** Diese Wartung wird zur Verbesserung des Systems angewendet. Dazu gehören das Erhöhen der Leistung, das Hinzufügen von neuen Features oder das Verbessern der Dokumentation.

- Preventive maintenance: Bei Tätigkeiten zur Verbesserung der Wartbarkeit und zur Vermeidung von Problemen in der Zukunft ist von Preventive maintenance die Rede. Hier werden zum Beispiel interne Abhängigkeiten neu geordnet. Dies dient zur Verbesserung der Kohäsion und der Kopplung.

Ein Großteil der Kosten, die im Softwarelebenszyklus aufkommen, entstehen durch die Wartung des Projekts. Die Wartung gilt auch als schwerste Phase, weil oftmals der Code beispielsweise schlecht strukturiert oder schwer zu verändern ist. [YL94] [SM98] Laut Chen et al. [CH09] sind die typischen Faktoren, die zu Problemen bei der Wartung des Softwaresystems führen, folgende:

- Qualitätsprobleme in der Dokumentation: Lientz und Swanson fanden heraus, dass mangelnde Qualität bei der Dokumentation einer der Hauptgründe für Probleme in der Wartungsphase ist [LS81]. Dokumentationen sind wichtig für das Verständnis und die Veränderbarkeit von Software. Wenn die Qualität der Dokumente gering ist oder Dokumente fehlen, führt dies zu großen Problemen in der Implementierung und später auch bei der Wartung von Software. Oftmals werden die Dokumente angelegt und in den späteren Phasen der Softwareentwicklung nicht mehr oder nur mangelhaft angepasst.
- Qualitätsprobleme im Code: Eine niedrige Qualität des Programmcodes führt zu Wartbarkeitsproblemen. [LS81] Auch ein hohes Level an Komplexität hat negative Auswirkungen auf die Wartung. [GK91] Dementsprechend wurden bereits einige Methoden zur Verbesserung der Codequalität vorgeschlagen, wie zum Beispiel die Aufteilung der Software in kleinere Module und Komponenten, das Benutzen von Kommentaren und das Durchführen von Refactoring. Refactoring verbessert die Struktur des Programms, ohne jedoch das Programmverhalten zu verändern. Das Programm soll einfach lesbar, verständlich und modifizierbar sein. [FBB+99]
- Probleme bei der Beschreibung der Systemanforderungen: Die meisten Mängel in Softwaresystemen können auf Probleme, die während der Anforderungsphase aufgetreten sind, zurückgeführt werden. Studien haben gezeigt, dass der Großteil der Fehler durch falsche, unvollständige, missverständliche oder unklare Systemanforderungen zustande kommt. [Mog01] Außerdem führen auch unrealistische, widersprüchliche oder sich immer ändernde Systemanforderungen zu einer schlechteren Qualität des Softwareprodukts. [Nid96]
- Probleme beim Personal und den vorhandenen Ressourcen: Es wurde herausgefunden, dass fehlende Ressourcen und fehlende Zeit große Auswirkungen auf die Softwarequalität haben. [WH98] Viele Studien zeigen außerdem, dass personelle Faktoren wie häufiges Wechseln von Teammitgliedern, unzureichende Fähigkeiten und Kenntnisse, fehlende Erfahrung und mangelnder Einsatz im Projekt oft zum Scheitern des Projektes geführt haben.
- Probleme durch schlechtes Geschäftsprozessmanagement: In Studien wurde gezeigt, dass fehlende Unterstützung durch das Prozessmanagement zu Risiken im Projekt und zum Scheitern des Projekts führen können. [JK99] Bei den gescheiterten Projekten wurden sechs häufig auftretende Probleme festgestellt: mangelnde Projektplanung, schlechte Kostenschätzung, Nicht-Einhaltung von Meilensteinen, fehlende Änderungskontrolle, schlechte Messungen/Analysen und fehlende Qualitätskontrolle. [Jon04] Außerdem war bei vielen Projekten die Qualitätsuntersuchung zur Sicherstellung des Qualitätsniveaus ineffektiv. [Gal03]

Um diese Problemfaktoren anzusprechen, muss auf eine gute Wartbarkeit der Software geachtet werden. Die Wartbarkeit ist ein wichtiges Software-Qualitätsattribut und kann bei guter Umsetzung die Softwarekosten, die in der Phase der Softwarewartung entstehen, erheblich senken. Die Definition von Wartbarkeit ist laut dem IEEE Glossar folgende:

„Wartbarkeit ist die Einfachheit, mit der ein Softwaresystem oder eine Softwarekomponente modifiziert werden kann, um Fehler zu beseitigen, die Leistung oder andere Attribute zu verbessern oder sich an veränderte Umgebungen anzupassen.“ [RGK90]

Dabei gibt es keine feststehenden Kriterien, wie Wartbarkeit zu messen und zu bewerten ist. Es gibt bereits viele Forschungsarbeiten, die ihre eigenen Methoden zur Evaluation von Wartbarkeit präsentieren, jedoch sind diese oft inkonsistent.

2.4.1 Qualitätsmodelle für die Wartbarkeit

Hashim und Key [Has96] präsentierten zum Beispiel ein Wartungsmodell mit einer Reihe von Attributen, nach denen die Wartbarkeit eines Softwaresystems untersucht werden kann. Sie stellten folgende Hauptfaktoren vor:

- **Modularität:** Modulare Systeme sind von Vorteil, weil sie einfacher zu verstehen und zu erklären sind. Dadurch wird auch die Dokumentation erleichtert. Außerdem können Gruppen unabhängig voneinander an dem Projekt arbeiten, da jede Gruppe ein anderes Modul implementieren kann. Das System ist außerdem leichter wartbar, da Änderungen in einem Modul vorgenommen werden können, ohne dass der Rest des Systems davon beeinflusst wird. Zudem können die einzelnen Module separat getestet und integriert werden.
- **Lesbarkeit:** Lesbarkeit beschreibt wie schnell und einfach der Leser den Programmcode versteht. Dieser Aspekt ist wichtig, da jedes Programm in allen Softwareentwicklungsphasen mehrmals gelesen wird und eine schlechte Lesbarkeit die Arbeit unnötig verzögert. Die Lesbarkeit des Programms kann durch eine saubere Dokumentation verbessert werden.
- **Wahl der Programmiersprache:** Ist der Code in einer Programmiersprache geschrieben, mit der viele Entwickler einer Organisation nicht vertraut sind, erschwert dies die Wartung. Außerdem sind high-level Programmiersprachen meist einfacher zu verstehen als low-level Sprachen.
- **Standardisierung:** Eine Anleitung mit einer Reihe an Programmierstandards, an die sich alle Mitwirkenden beim Schreiben des Codes halten sollen, sorgt dafür, dass der Code einheitlicher ist und später schneller von anderen Personen verstanden werden kann.
- **Level der Validierung und Tests:** Es soll mehr Zeit und Arbeit für die Überprüfung des Designs und das Testen des Programmcodes verwendet werden, sodass weniger Fehler im Programm auftreten.
- **Komplexität:** Die Komplexität der Software beeinflusst die Lesbarkeit und damit auch die Wartbarkeit des Codes. Die Komplexität hängt dabei vom Kontrollfluss des Moduls, vom Datenfluss und von der benutzten Datenstruktur ab.

- **Rückverfolgbarkeit:** Rückverfolgbarkeit bedeutet, dass eine Repräsentation des Designs oder aktuelle Programmkomponenten auf ihre Anforderungen zurückverfolgt werden können. Dafür sollen Informationen vorhanden sein, womit diese Rückverfolgung durchgeführt werden kann.

Ein anderer Ansatz zur Charakterisierung von Wartbarkeit wurde von der Software Improvement Group [BCSV12] vorgestellt. Sie fokussiert sich auf die technisch analysierbaren Elemente des Programmcodes. Dafür wurden die folgenden sechs Hauptaspekte für die Bewertung der Qualität ausgewählt, mit denen der Code untersucht werden kann:

- **Größe:** Je größer das System ist, desto schwerer wartbar ist es, da mehr Informationen untersucht werden müssen.
- **Redundanz:** Der gleiche Code sollte nicht an mehreren Stellen auftreten.
- **Größe der Unit:** Units sollen klein gehalten werden, damit diese fokussiert bleiben und einfacher zu verstehen sind.
- **Komplexität:** Die Systeme sollen schlicht gehalten sein, da sie so verständlicher sind als komplexe Systeme.
- **Schnittstellengröße Unit:** Die Units sollen nicht zu viele Parameter enthalten, da dies ein Symptom für schlechte Datenkapselung darstellt.
- **Kopplung:** Die Komponenten sollen nicht eng gekoppelt sein, da eng gekoppelte Komponenten durch die hohe Bindung schwer zu verändern sind.

2.4.2 Wartbarkeit von Service-orientierten Systemen

Um qualitativ hochwertige SOA-basierte Systeme zu entwickeln, muss von Anfang an auf eine gute Systemqualität hingearbeitet werden. Die Anwendung von explizitem Qualitätsmanagement in den Anfangsstadien des Projekts vermeidet das Auftreten von Problemen, die in den späteren Stadien des Softwarelebenszyklus adressiert werden müssen.

Jedoch können die im vorherigen Kapitel genannten Qualitätsmodelle nicht direkt auf SOA-basierte Systeme angewandt werden. Der Grund liegt in den inhärenten Unterschieden zwischen Service-orientierten Systemen und objektorientierten bzw. prozeduralen Systemen. Beispielsweise existieren bei objektorientierten Systemen die Konzepte Klassen, Methoden, Attribute und Vererbung. Diese Konzepte gibt es jedoch nicht bei Service-orientierten Systemen. Somit stimmen die Systemkomponenten nicht überein und es können keine OO-Metriken angewandt werden, ohne diese explizit für SOA-Systeme anzupassen. [SCKP08]

Senivongse und Puapolthep [SP15] haben ein Qualitätsmodell entworfen, mit dem die Wartbarkeit für Service-orientierte Systeme bewertet werden kann. Dieses Modell ist in vier Level unterteilt. Level 1 beinhaltet die Qualitätsattribute des Systems. Diese Qualitätsattribute werden in high-level (1H), medium-level (1M) und low-level (1L) aufgeteilt. Level 2 besteht aus den Service-Systemeigenschaften, diese beschreiben die Qualitätsattribute von 1L. In Level 3 sind die Metriken enthalten, mit denen die Service-Systemeigenschaften objektiv gemessen werden können. Level 4 beinhaltet die Systemkomponenten der Services, die von den Metriken benutzt werden können, wie z.B. Services, Schnittstellen, Nachrichten und Abhängigkeiten zwischen Systemkomponenten.

Die Wartbarkeit eines Systems ist dabei eines der vielen high-level Qualitätsattribute eines Softwaresystems. Die Wartbarkeit setzt sich aus den medium-level Attributen Analysierbarkeit, Veränderbarkeit, Stabilität und Testbarkeit zusammen.

Analysierbarkeit charakterisiert die Fähigkeit, den Grund eines Softwareausfalls oder die fehlerhaften Punkte eines Softwaresystems mithilfe einer Analyse zu finden. Sie beinhaltet die folgenden low-level Attribute:

- Die **Lesbarkeit** ist abhängig vom Programmierstil und bezieht sich darauf, wie einfach der Service zu lesen ist. Die Lesbarkeit kann durch die Service-Systemeigenschaft *Lesbarkeitslevel* beschrieben werden. Das *Lesbarkeitslevel* kann durch die Metrik *Total readability* berechnet werden.
- Das Attribut **Verständlichkeit** bezieht sich auf die Anstrengung, die nötig ist, das Design des Systems zu lernen und zu verstehen. Dieses Attribut hängt von einer Vielzahl an Service-Systemeigenschaften ab: *Kopplung*, *Kohäsion*, *Komplexität*, *Systemgröße*, *Service Granularität*, *Parameter Granularität* und *Konsumierbarkeit*. Metriken hierfür sind zum Beispiel: *Average number of directly connected services* für die Messung der *Kopplung* oder *Number of operations* für die Messung der *Komplexität*.
- Die **Erreichbarkeit** charakterisiert, wie einfach der Service während der Entwicklungs- und Wartungsphase erreicht werden kann. Die Erreichbarkeit hängt von der Service-Systemeigenschaft *Erreichbarkeitslevel* ab, welche durch die Metrik *Total accessibility* berechnet werden kann.

Veränderbarkeit beschreibt das Ausmaß des Aufwands, der für die Veränderung eines Systems benötigt wird. Folgende low-level Attribute umfasst die Veränderbarkeit:

- Die **Kopplungsstruktur** bezieht sich auf die Stärke der Abhängigkeiten zwischen Services. Die Service-Systemeigenschaft hierfür ist die *Kopplung*, diese wird durch die Metrik *Average number of directly connected services* analysiert.
- Die **Isolierbarkeit** hängt von der Stärke der Beziehung zwischen den Operationen eines Dienstes ab. Die Isolierbarkeit wird von der Service-Systemeigenschaft *Kohäsion* bestimmt. Die Metrik zur Berechnung dieser Eigenschaft ist *Inverse of average number of used message*.
- Das Attribut **funktionale Abdeckung** bezieht sich auf die Größe und Abdeckung der Funktionen eines Dienstes und hängt von der Anzahl der Daten, die mit einem Dienst ausgetauscht werden, ab. Die Service-Systemeigenschaften für die funktionale Abdeckung sind *Service Granularität* und *Parameter Granularität*. Die Metrik für die Untersuchung von *Service Granularität* ist *Squared average number of operations to squared average number of messages*. Die Metrik für *Parameter Granularität* heißt *Coarse-grained parameter ratio*.

Stabilität charakterisiert die negativen Auswirkungen, die durch Systemänderungen ausgelöst werden. Ein gutes System soll bei Veränderungen trotzdem seine Dienste weiter anbieten. Die Stabilität enthält folgende low-level Attribute:

- Die **Verfügbarkeit** bezieht sich auf die Häufigkeit und Länge der Ausfallzeit, wenn der Service Wartungen und Änderungen am System durchführt. Das Attribut wird durch die Service-Systemeigenschaft *Verfügbarkeitslevel* beschrieben, die dazugehörige Metrik heißt *Total availability*.

- Die **Datenkapselung** hängt von der Größe der Daten ab, die mit einem Service ausgetauscht werden. Die zugehörige Service-Systemeigenschaft ist wieder die *Parameter Granularität* und wird mit der Metrik *Coarse-grained parameter ratio* untersucht.
- Das Attribut **Unabhängigkeit der Veränderungen** bezieht sich auf das Ziel, Änderungen zu vermeiden, die andere Teile des Systems beeinflussen würden. Dieses Attribut wird durch die Service-Systemeigenschaft *Level der Unabhängigkeit von Änderungen* beschrieben und kann durch die Metrik *Total independence of changes* untersucht werden.

Testbarkeit ist das Ausmaß des Aufwands, der nötig ist, um eine Systemänderung zu verifizieren und zu testen. Für die Testbarkeit sind folgende low-level Attribute wichtig:

- Der **Fokus auf Umgebung** beschreibt, dass beim Testen des Systems der Fokus auf die Serviceumgebung gerichtet sein soll. Die Service-Systemeigenschaft ist hier das *Umgebungs-fokuslevel* und wird durch die Metrik *Total environment focus* analysiert.
- Das Attribut **Prozesssimulation** besagt, dass nicht nur die Services und die Serviceumgebung getestet werden sollen, sondern auch der Prozess. Die Prozesssimulation wird durch die Service-Systemeigenschaft *Prozesssimulationslevel* beschrieben. Die Metrik *Total process simulation* berechnet das *Prozesssimulationslevel*.

Um nun Werte für die medium-level und high-level Qualitätsattribute zu bekommen, werden die Werte der Metriken des niedrigeren Levels erst durch die Anzahl der verschiedenen Metriken dividiert und die daraus entstandenen Werte dann addiert. Beispielsweise wird die Wartbarkeit folgendermaßen berechnet:

$0,25 * \text{Analysierbarkeit} + 0,25 * \text{Änderbarkeit} + 0,25 * \text{Stabilität} + 0,25 * \text{Testbarkeit}$.

Bogner et al. [BWZ17] haben in einer Literaturanalyse weitere relevante Metriken für die Wartbarkeit von Service-orientierten Systeme präsentiert. Diese Metriken fokussieren sich auf die Qualitäts-Eigenschaften Größe, Komplexität, Kopplung und Kohäsion.

Die **Größe** bezieht sich auf die Größe des Gesamtsystems, also die Summe der Größen der darunterliegenden Services. Ein größeres System hat negative Auswirkungen auf die Wartbarkeit. Hier gibt es die Metrik *Weighted Service Interface Count* (WSIC). WSIC beschreibt die Menge aller offenliegenden Schnittstellenoperationen eines Dienstes. Diese Operationen sind basierend auf der Granularität oder der Anzahl der Parameter unterschiedlich gewichtet. Niedrigere WSIC-Werte haben einen positiven Einfluss auf die Wartbarkeit.

Die **Komplexität** bezieht sich auf den Umfang und die Vielfalt der ausgeführten internen Arbeiten durch ein System, sowie auf den Grad der Interaktion zwischen seinen Services, der notwendig ist, um diese Arbeiten durchzuführen. Hohe Komplexität hat negative Auswirkungen auf die Wartbarkeit.

Eine Metrik zur Untersuchung der Komplexität ist *Total Response for Service* (TRS). Die dafür benötigte Metrik *Response for Operation* (RFO) ist die Anzahl der Sequenzen der anderen Operationen und Implementierungselemente, die bei einer einkommenden Anfrage für die untersuchte Operation durchgeführt werden können. TRS ist die Summe aller RFOs in der Serviceschnittstelle. Es gilt: ein niedriger TRS-Wert deutet auf eine niedrige Komplexität hin.

Eine weitere beschriebene Metrik ist *Number of Versions per Service* (NVS). Sie enthält die Anzahl an Versionen über die totale Anzahl von Services. Ein großer NVS-Wert impliziert eine hohe Komplexität.

Kopplung ist der Grad der Stärke von Interdependenzen und Verbindungen von einem Service zu anderen Services. Lose gekoppelte Systeme sind besser für die Wartbarkeit.

Im Rahmen der Kopplung wurde die Metrik *Service Interdependence in the System* (SIY) vorgestellt. Diese Metrik untersucht die Anzahl an Servicepaaren, die bi-direktional voneinander abhängig sind. Der ideale Wert ist hier 0.

Eine weitere Metrik für die Kopplung ist *Absolute Importance of the Service* (AIS). Diese Metrik bezieht sich auf die Anzahl der Konsumenten, die von dem untersuchten Service abhängig sind. Das sind Clients, die mindestens eine Operation vom Service aufgerufen haben. Mit dieser Metrik können die wichtigen Dienste des Systems herausgefunden werden.

Kohäsion bezieht sich auf das Ausmaß, in dem die Operation eines Services nur einer einzelnen Aufgabe oder Funktionalität beiträgt. Ein hoher Grad an Kohäsion hat positive Auswirkungen auf die Wartbarkeit.

Die erste vorgestellte Metrik hierfür ist *Service Interface Data Cohesion* (SIDC). Diese Metrik berechnet die Kohäsion anhand der Ähnlichkeiten des Parameterdatentyps und des Rückgabedatentyps der Operationen einer untersuchten Service-Schnittstelle. Ein Service wird als kohäsiv angesehen, wenn alle möglichen Paare von Operationen mindestens einen gleichen Parameterdatentyp und einen gleichen Rückgabedatentyp haben. Die Werte von SIDC gehen von 0 bis 1, wobei 1 die stärkste Kohäsion bedeutet. [PRF07]

Service Interface Usage Cohesion (SIUC) ist eine weitere Metrik für die Untersuchung der Kohäsion. SIUC ist definiert als das Verhältnis zwischen der addierten Anzahl der benutzten Operationen pro Client und der Anzahl der Clients multipliziert mit der Anzahl der Operationen in der Serviceschnittstelle. Die Werte von SIUC gehen von 0 bis 1, wobei 1 wieder die stärkste Kohäsion bedeutet.

3 Verwandte Arbeiten

Versuche, die Qualität eines Webservices zu bewerten, existieren schon lange. Beispielsweise haben Tsai et al. bereits im Jahr 2002 in ihrem Forschungsbericht ein XML-basiertes objektorientiertes Test Framework für Webservices [TPSC02] vorgestellt. Mit diesem Framework konnten selbstdefinierte Testfälle an WSDL Dateien durchgeführt werden. Das Problem hierbei war jedoch, dass die Tests trotzdem manuell von Menschen durchgeführt werden mussten. Bartolini et al. stellten daher in ihrem wissenschaftlichen Bericht [BBMP08] ein Framework vor, welches das auf WSDL Spezifikationen basierende Testen weiter automatisiert. Sie präsentierten eine Methode, wie automatisch Testnachrichten von WSDL Beschreibungen abgeleitet werden können. Aus diesen Testnachrichten konnte eine sofort nutzbare Testsuite gebaut werden, die sämtliche Webservice Operationen abdeckt und datengetriebene Testfälle anwendet.

In beiden Arbeiten wird ausschließlich die Funktionalität getestet, diese ist jedoch nicht das einzige Qualitätsattribut. Auch die Wartbarkeit ist ein wichtiges Qualitätsattribut. Der Fokus dieser Arbeit liegt auf der Untersuchung der Wartbarkeit und der generellen strukturellen Qualität der REST API. Dies geschieht durch die Analyse von OpenAPI Spezifikationen mithilfe von Metriken. Im Folgenden werden wissenschaftliche Arbeiten vorgestellt, die anhand von Metriken eine strukturelle Analyse der Spezifikationen von Webservices durchführen und dadurch Informationen über deren Qualität gewinnen.

3.1 Analyse von SOAP-basierten Webservices

Misra und Baski haben in ihrem wissenschaftlichen Artikel „Metrics suite for maintainability of eXtensible Markup Language web services“ [BM11] eine Testsuite zur Bewertung der Qualität von Webservices in Bezug auf die Wartbarkeit vorgestellt. Hierfür stellten sie eine Reihe von Metriken vor, die lediglich das WSDL Dokument als Quelle benötigen:

- **Data Weight (DW):** Mithilfe dieser Metrik kann die strukturelle Komplexität von Datentypen aus Service-Nachrichten berechnet werden. Sie kann auch als Indikator dafür angesehen werden, wie groß der Aufwand für den Nutzer ist, die Nachricht zu verstehen.
- **Distinct Message Ratio (DMR):** Misra und Baski haben herausgefunden, dass einige Nachrichten die gleiche Struktur haben und damit auch den gleichen DW-Wert. Die gleiche Struktur führt zu mehr Vertrautheit und damit nimmt auch der Aufwand für das Lesen der Nachricht ab. DMR erfasst genau diesen Aspekt des verringerten Aufwands durch Nachrichten mit gleicher Struktur.

- **Message Entropy (ME):** Diese Metrik bildet die Prozentzahl der ähnlich-strukturierten Nachrichten in einem WSDL Dokument ab. Sie bezieht sich außerdem auch auf Situationen, in denen mehrere gleich strukturierte Nachrichten nacheinander auftreten. Ein niedriger ME-Wert sagt aus, dass die Nachrichten strukturell konsistent sind, wodurch die Komplexität des Dokuments sinkt.
- **Message Repetition (MRS):** Diese Metrik analysiert die Vielfalt an Strukturen und damit die Daten-Komplexität von WSDL Dokumenten. Höhere Werte bedeuten weniger Komplexität und somit auch weniger Aufwand für den Entwickler, sich auf die Nachrichtenstruktur zu konzentrieren, da die Struktur sich öfter wiederholt und daher bereits bekannt ist.

Auf den Artikel von Misra und Baski aufbauend, haben Mateos et al. in ihrer Arbeit „Managing Web Service Interface Complexity via an OO Metric-based Early Approach“ eine Methode vorgestellt, mit der die Komplexität der Service-Schnittstelle verringert werden kann. Da Service-Schnittstellen standardmäßig mit der Code First Methode gebaut werden, hat das Refactoring der Implementierung eine positive Auswirkung auf das WSDL Dokument. Dies ist insbesondere der Fall, wenn die Generierung des WSDL Dokuments automatisch aus dem Code erfolgt. Die Verbesserungen, die durch das Refactoring des Codes entstehen, werden somit direkt auf das WSDL Dokument übertragen.

Mateos et al. haben eine Reihe von Metriken auf OO-Ebene ausgewählt und untersucht, welchen Einfluss diese auf die vorher beschriebenen Komplexitäts-Metriken haben. Hierfür wurden zwei Experimente durchgeführt, für die Refactoring Methoden verwendet wurden, die die OO-Metriken *Coupling Between Objects* (CBO) und *Weighted Method per Class* (WMC) beeinflusst haben. *Coupling Between Objects* zählt, wie viele Methoden und Instanzvariablen von anderen Klassen von der untersuchten Klasse aufgerufen werden. Je mehr Methoden und Instanzvariablen aufgerufen werden, desto höher ist die Kopplung zwischen den Klassen. *Weighted Method per Class* zählt die Anzahl der Methoden einer Klasse. Je größer die Zahl ist, desto höher ist die Wahrscheinlichkeit, dass ein Paar von Methoden nicht zusammenhängen und somit über eine schwache Kohäsion verfügen. [OMCZ13]

Beim ersten Experiment wurde das Refactoring des Codes in drei Phasen mit zwei Refactoring-Methoden nach Fowler's Katalog [FBB+99] durchgeführt:

- In der ersten Refactoring Phase wurde das Gegenteil von REPLACE DATA VALUE WITH OBJECT angewendet. Objekte wurden durch primitive Datentypen ersetzt, wodurch die Komplexität der Nachricht reduziert wird. Dadurch verringert sich auch der Wert von *Coupling Between Objects*.
- Die zweite Phase erfolgte mithilfe der Refactoring-Methode MOVE METHOD. Hier wurden Methoden von Klassen, die Service-Schnittstellen beschreiben, ausgewählt und Teile der Methoden in ein neues Interface verschoben. Durchgeführt wurde dies bei Klassen, die eine übermäßig große Menge an Methoden besitzen. Durch das Verschieben wurde der Wert von *Weighted Method per Class* für die Klasse reduziert.
- In der dritten Phase wurde die Methode MOVE METHOD aus Phase 2 auf den bereits refaktorierten Code aus Phase 1 angewandt. Dadurch können die Auswirkungen auf die Komplexitätsmetriken überprüft werden, wenn simultan die Metriken CBO und WMC verändert werden.

Die Auswirkungen auf die Komplexitäts-Metriken wurden daraufhin getestet, indem ein WSDL Dokument aus dem refaktorierten Code mithilfe von Werkzeugen wie Java2WSDL generiert wird. Das Ergebnis des Experiments war, dass alle drei Phasen des Refactorings Auswirkungen auf die Komplexitätsmetrik *Data Weight* (DW) hatten, ihr Wert ist bei allen Phasen gesunken. Die Metriken *Distinct Message Ratio*, *Message Entropy* und *Message Repetition* haben sich jedoch nur in der ersten Phase des Refactorings verbessert. Daraus folgend wurde die Vermutung aufgestellt, dass lediglich die OO-Metrik CBO positive Auswirkungen auf die drei genannten Metriken hat und WMC nicht. Um die Korrelation zwischen WMC und den Metriken DMR, ME und MRS genauer zu untersuchen, wurde ein weiteres Experiment durchgeführt.

In diesem zweiten Experiment wurden bestimmte Grenzwerte für die maximale Anzahl an Operationen in einem Service ausgewählt. Daraufhin wurde eine Klasse, die eine Service-Schnittstelle implementiert hat, fragmentiert, sodass aus dieser Klasse neue Services generiert wurden, die maximal so viele Operationen besitzen wie der Grenzwert groß ist. Dadurch besteht eine direkte Abhängigkeit zwischen dem gewählten Grenzwert und dem Wert der Metrik *Weighted Method per Class*. Das Ergebnis dieses Experiments ist, dass sich die Vertrautheit des Services verringert hat. Durch die reduzierte Anzahl an Operationen ist die Anzahl der ähnlich-strukturierten Operationen kleiner geworden. Daher haben sich auch die Ergebnisse der Metriken DMR und MRS verschlechtert. Im Gegensatz dazu hat sich die Konsistenz der Nachrichtenstruktur verbessert, weil die Anzahl der Operationen gesunken ist. Dies hatte positive Auswirkungen auf die ME-Metrik.

Eine weitere verwandte Arbeit wurde von Sneed [Sne10] vorgestellt. Hier lag der Fokus auf der Größe, der Komplexität und der Qualität von Schnittstellen. Dafür wurde ein System zur statischen Analyse dieser Qualitätsattribute anhand von WSDL Dokumenten entwickelt.

Die Größe der Schnittstelle bezieht sich auf die Anzahl der Zeilen im Code, den Variablen, den Operationen und den Eingaben und Ausgaben. Hier wurden die Variablen in zwei unterschiedliche Typen aufgeteilt: *elementare Variablen* wie zum Beispiel Strings oder Integers und *Gruppenvariablen*, die elementare Variablen und weitere komplexe Datentypen beinhalten können.

Für die Komplexität wurde die *linguale* und die *strukturelle* Komplexität berechnet und der Median der beiden gebildet. Die linguale Komplexität bezieht sich auf die Anzahl der unterschiedlichen semantischen Elementtypen, die in einem Text benutzt werden, verglichen mit der Gesamtzahl des Vorkommens semantischer Elemente. Die strukturelle Komplexität hängt von der Anzahl der Entitätsbeziehungen und der Entitätstypen ab. Je mehr Beziehungen es im Verhältnis zur Anzahl der Entitätstypen gibt, desto höher ist auch die strukturelle Komplexität.

Die Qualität der Schnittstelle wurde in die Attribute *Wiederverwendbarkeit*, *Testbarkeit*, *Modularität* und *Wartbarkeit* unterteilt. Die Wiederverwendbarkeit hängt dabei von der Anzahl der extern referenzierten Datentypen ab. Je weniger externe Beziehungen existieren, desto höher ist die Wiederverwendbarkeit. Die Testbarkeit wird durch die Anzahl der zu testenden Operationen und deren Anzahl an Parametern bestimmt. Je weniger Operationen getestet werden müssen und je weniger Parameter diese Operationen haben, desto besser ist es für die Testbarkeit. Modularität hängt von der Kopplung und Kohäsion der Operationen ab und ist der Durchschnitt der beiden Werte. Unter Wartbarkeit wird in dieser Arbeit die Einhaltung der Regeln für die Entwicklung wartbarer Webservice-Schnittstellen verstanden. Die Metrik hierfür heißt *Konformität* und wird berechnet aus den Gewichten der Regelverletzungen durch die Anzahl der WSDL Anweisungen. Regelverletzungen mit hoher Signifikanz haben dabei ein höheres Gewicht.

Dem Softwaresystem wird eine Liste von WSDL Dokumenten und eine Reihe an Regeln, die überprüft werden sollen, übergeben. Außerdem bekommt das System eine Liste von Namensvorlagen. Das WSDL Dokument wird nun geparsed, die beinhalteten Namen mithilfe der Namensvorlage überprüft und die verschiedenen Qualitätsattribute analysiert. Daraufhin wird ein Mangelbericht mit den Regelverletzungen und ein Metrik-Bericht ausgegeben.

3.2 Analyse von RESTful APIs

Da RESTful API eine relativ neue Technologie ist, gibt es im Gegensatz zu Analysen von WSDL Spezifikationen wenige wissenschaftliche Arbeiten über die Analyse von RESTful APIs auf Grundlage von API Beschreibungsformaten. Einer der ersten Ansätze zur Analyse von RESTful APIs mithilfe von Spezifikationen stammt von Haupt et al. [HLV18].

Sie stellten ein Framework für die automatische Analyse einer REST API anhand ihrer maschinenlesbaren API Beschreibungen auf. Der Fokus der Analyse liegt auf der Struktur der API und kann bereits vor der konkreten Implementierung durchgeführt werden. Die Ergebnisse der Analyse werden in einer CSV-Datei gespeichert und können über einen von den Autoren entwickelten grafischen Editor visualisiert werden. Die API Beschreibungen werden auf ihre strukturellen Aspekte reduziert und ein Metamodell daraus gebaut, sodass keine für die Analyse irrelevanten Informationen verarbeitet werden müssen. Als API-Beschreibungsformate wurden wegen ihrer Popularität Swagger und RAML gewählt. Die Metamodelle der beiden Spezifikationen sind nicht identisch, was zu Inkonsistenzen und Fehlern bei der Analyse führen kann. Um das zu verhindern, haben die Autoren beschlossen ein eigenes kanonisches Metamodell für REST APIs zu entwickeln.

Auf diese kanonischen Metamodelle werden dann Metriken angewandt. Haupt et al. führten einen Testlauf mit 286 Swagger Spezifikationsdokumenten, unter anderem von Azure, Google und Github, durch. In Abbildung 3.1 werden alle untersuchten Metriken inklusive ihrer Ergebnisse aufgelistet. Die Ergebnisse umfassen hier den kleinsten Wert, den größten Wert, den Durchschnitt und den Median.

	Min	Max	Mean	Median
#Resources	1	264	20.3	9
#ReadOnly resources	0	227	10.4	4
#POST	0	93	6.5	3
#DELETE	0	40	2.6	1
#Roots	1	227	8.1	4
#Links	0	248	12.2	4
MaxDepth	0	7	1.8	1
#Components	1	227	8.1	4
Smallest component	1	165	2.4	1
Biggest component	1	165	8.3	3.5
Avg component size	1	165	4.0	2
Biggest component coverage	0.4%	100.0%	54.0%	50.0%

Abbildung 3.1: Überblick der untersuchten Metriken [HLV18]

Die ersten vier Metriken beziehen sich auf Ressourcen. Beim Vergleichen der Ergebnisse der vier Metriken ist erkennbar, dass *read-only* Ressourcen bei REST APIs sehr beliebt sind und im Durchschnitt die Hälfte der Ressourcen ausmachen. Außerdem gibt es mehr Ressourcen mit *POST*-Methoden als Ressourcen mit *DELETE*-Methoden, was hauptsächlich an den sogenannten *command*-Ressourcen liegt. Die *command*-Ressourcen repräsentieren eine Funktionalität und werden standardmäßig mit der *POST*-Methode aufgerufen.

Die nächsten drei Metriken beziehen sich auf die Verbindungen zwischen Ressourcen. Hier ist erkennbar, dass einige der APIs eine Vielzahl an *Root*-Ressourcen haben, was in manchen Richtlinien als negativ angesehen wird. Außerdem ist der Durchschnittswert der Metrik *MaxDepth* ziemlich gering, was bedeutet, dass es wenige Möglichkeiten gibt, tiefer in die API zu navigieren, aber dafür mehr Alternativen existieren, um auf einer Ebene weiter zu navigieren.

Die letzten fünf Metriken beziehen sich auf die Komponenten. Die Metrik *Biggest component coverage* beschreibt den Anteil der API, der von der größten Komponente abgedeckt wird. Wenn diese Metrik einen hohen Wert hat, bedeutet das, dass von einer einzigen *Root*-Ressource zu den meisten anderen Ressourcen gelangt werden kann, was wiederum auf eine bessere Struktur der API hindeutet. In der Analyse kam heraus, dass 22% der analysierten APIs eine *Biggest component coverage* von über 90% besitzen und damit auch als gut strukturiert angesehen werden können.

Aufbauend auf den oben genannten Metriken stellten Haupt et al. eine neue Metrik mit dem Namen „user-perceived complexity of an API“ vor. Die Metrik beschreibt, wie hoch ein Nutzer die Komplexität einer API empfindet, ohne sie objektiv zu messen. Ist die empfundene Komplexität hoch, kann dies die Verwendung und Verbreitung der API negativ beeinflussen. Das ist besonders dann schlecht, wenn die API öffentlich angeboten wird.

Für diese neue Metrik haben die Autoren eine Studie mit erfahrenen Softwareentwicklern durchgeführt, die grafische Repräsentationen von APIs paarweise im Hinblick auf die empfundene Komplexität vergleichen sollten. Im nächsten Schritt wurden aus zwei Metriken der Abbildung 3.1 verschiedene Aggregationen erstellt und auf die Relation zur wahrgenommenen Komplexität untersucht. Es wurde herausgefunden, dass manche der aggregierten Metriken die *user-perceived complexity of an API* der untersuchten APIs besser beschreiben als die Metriken einzeln.

Eine weitere wissenschaftliche Arbeit zur Analyse von RESTful APIs wurde von Kulkani [KT18] vorgestellt. Hier wurde ein Softwaresystem vorgestellt, das überprüft, ob eine Implementierung korrekt nach den Anforderungen der API Spezifikation entwickelt wurde.

Das System bekommt als Eingaben ein manuell geschriebenes Swagger 2.0 Dokument und ein weiteres Dokument im Swagger 2.0 Format, welches vom *Springfox Framework* aus der Implementierung generiert wurde. Die Software besteht aus zwei Modulen: das erste Modul ist für die Generierung des Dokuments aus dem Code zuständig, während das zweite das generierte Dokument mit dem Swagger Dokument vergleicht und die Unterschiede analysiert. Dabei untersucht das System die korrekt implementierten APIs, die fehlerhaften APIs, die nicht implementierten APIs und die implementierten, aber nicht dokumentierten APIs.

Es werden zwei Analyseberichte ausgegeben. Der erste Bericht gibt eine Übersicht darüber, ob die Services gemäß den Spezifikationsanforderungen implementiert wurden. Dabei werden folgende Punkte untersucht und in dem Bericht wiedergegeben: *Anzahl der Ressourcen*, *Anzahl der erwarteten Endpunkte*, *Anzahl der implementierten Endpunkte*, *Nicht erwartete, aber implementierte Endpunkte*, *Anzahl der Lücken im System* und *Prozentzahl des Systems vervollständigt*.

Der zweite Bericht untersucht jeden dokumentierten Endpunkt und vergleicht diesen mit den implementierten Endpunkten. Es werden folgende Details über die Endpunkte analysiert und in dem

Bericht gespeichert: *Erwarteter Parameter und implementierter Parameter, Erwarteter „consumes“ MIME-Type und implementierter „consumes“ MIME-Type, Erwarteter „produces“ MIME-Type und implementierter „produces“ MIME-Type, Erwartete HTTP-Antworten und implementierte HTTP-Antworten.*

Durch die Ergebnisse der beiden Analyseberichte kann der Unterschied zwischen der erwarteten und der aktuellen Implementierung gezeigt werden. Nach Kulkani kann mithilfe dieser Herangehensweise auch untersucht werden, ob die Implementierung und die Dokumentation REST-Standards folgen und ob sie REST-konform sind.

4 Entwicklung eines OpenAPI Evaluationstools

Aufbauend auf den theoretischen Grundlagen und den verwandten Arbeiten, die in den vorigen Kapiteln beschrieben wurden, bestand die Hauptaufgabe dieser Bachelorarbeit darin, ein prototypisches Softwaresystem zu entwickeln, um die Wartbarkeit von Schnittstellen automatisch analysieren zu können.

Im Vergleich zu vielen anderen Arbeiten, die eine Schnittstelle lediglich auf generelle Eigenschaften hin untersuchen, liegt der Fokus bei diesem Projekt auf der gezielten Analyse der Wartbarkeit. Die bisherigen Softwaresysteme haben die Ergebnisse der Analyse meist lediglich in einem menschenlesbaren Dokument ausgegeben. In dieser Arbeit wurde zusätzlich besonderer Wert auf die Maschinenlesbarkeit gelegt. Dadurch können die Ergebnisse maschinell weiterverarbeitet werden, was so bisher nicht möglich war. Ein weiterer Unterschied zu vergangenen Arbeiten ist, dass für die Analyse OpenAPI Spezifikation in der Version 3 benutzt wird. Dies ist die erste Version seit der Gründung der OpenAPI Initiative und hat damit besondere Aktualität. Zudem wird die hier entwickelte Funktionalität als REST Webservice angeboten. Somit kann in der Zukunft einfach ein Frontend hinzugefügt und das System in einer Webanwendung genutzt werden.

Im Folgenden werden die Anforderungen an dieses Projekt und dessen Umsetzung erläutert. Daraufhin wird die Architektur dargestellt und die Funktionsweise anhand eines Anwendungsbeispiels verdeutlicht.

4.1 Anforderungen

Die Grundanforderung besteht darin, ein Softwaresystem zu entwickeln, welches die Qualität von RESTful APIs in Bezug auf die Wartbarkeit analysiert. Die Analyse soll anhand von OpenAPI Spezifikationen geschehen und wird mithilfe von bestimmten Metriken durchgeführt. Es können jedoch nicht alle Metriken, die die Wartbarkeit von RESTful APIs bewerten, auf die OpenAPI Spezifikation angewandt werden, da für manche Metriken umfangreichere Daten benötigt werden, die in der Spezifikation jedoch nicht vorhanden sind.

Eine weitere Anforderung ist, dass das System einfach erweiterbar sein soll, also modular aufgebaut sein muss. Insbesondere soll es unkompliziert möglich sein, neue Metriken hinzuzufügen, um somit die Qualität der Auswertung durch eine detailliertere Analyse oder neue Erkenntnisse ständig verbessern zu können.

Auf Basis dieser Anforderungen entstand folgende grobe Funktionsweise: Das System soll aus zwei eigenständigen Anwendungen bestehen: eine Kommandozeilenapplikation als Kernkomponente und eine Webanwendung, die auf dieser Kernkomponente aufbaut.

Die Kernkomponente bekommt über die Kommandozeile eine OpenAPI Spezifikationsdatei eingelesen. Diese Datei wird analysiert und ein Wartbarkeitsbericht mit den Ergebnissen der verschiedenen angewandten Metriken wird generiert. Wichtig ist, dass die Kernkomponente eine Applikation ist,

die auch alleinstehend benutzt werden kann.

Bei der Webanwendung wird die OpenAPI Spezifikation über eine HTTP-Anfrage eingelesen. Dabei greift die Webapplikation auf die Klassen und Methoden des Kernsystems zu, um die Funktionalität des Systems über einen RESTful Webservice und deren HTTP-Methoden anzubieten.

Da es sich bei diesem Projekt um ein prototypisches System handelt, liegt der Fokus der Webanwendung auf dem Backend. Das Design der Oberfläche wird darum vorerst vernachlässigt und kann in der Zukunft ergänzt werden.

Der durch das System generierte Wartbarkeitsbericht soll sowohl in einem menschenlesbaren als auch maschinenlesbaren, standardisierten Datenformat sein. Er enthält die Ergebnisse der auf die OpenAPI Spezifikation angewandten Metriken sowie generelle Informationen über die eingelesene Spezifikation. Die Berichte sollen in einer Datenbank gespeichert werden, sodass in der Zukunft verschiedene Berichte miteinander verglichen werden können. Das ermöglicht es dem Nutzer, Durchschnittswerte und Ausreißer zu finden.

Eine weitere Anforderung ist die Einbettung in einen Docker-Container¹. Ein Docker-Container ist ein leichtgewichtiges, ausführbares Softwarepaket, in dem die Anwendung sowie sämtliche benötigten Ressourcen enthalten sind. Der Docker-Container wird benötigt, um Probleme zu vermeiden, die durch unterschiedliche Softwareumgebungen entstehen können. Bereits bei abweichenden Betriebssystemen können Unregelmäßigkeiten bei der Anwendung des Programms auftreten, aber auch bei Unterschieden wie z.B. in der Netzwerk-Topologie oder beim Speicher. Ein Docker-Container verbessert somit die Portabilität des Softwaresystems. Zudem wird auch das Deployen und Managen des Systems vereinfacht.

4.2 Methodisches Vorgehen

Vor Beginn der Umsetzung des Projekts erfolgte eine ausführliche Auseinandersetzung mit dem Thema OpenAPI Spezifikation und deren Anwendungsszenarien (siehe Abschnitt 2.3). Besonderer Fokus lag hierbei auf der Struktur und den verschiedenen Elementen der Spezifikation, da diese für das Projekt von großer Bedeutung sind.

Zusätzlich wurde die OpenAPI Spezifikation mit anderen Schnittstellenbeschreibungen wie WSDL verglichen. Die verschiedenen Beschreibungen wurden auf Unterschiede und Gemeinsamkeiten, sowie Vor- und Nachteile hin untersucht, um die Charakteristika der Spezifikationen besser zu verstehen.

Außerdem wurde das Wissen über Wartbarkeit in Bezug auf Serviceorientierung erweitert. Es wurden verschiedene Metriken zur Messung von Wartbarkeit analysiert und für den Einsatz bei Service-orientierten Systemen geprüft. Voraussetzung dafür war, dass diese Metriken auf eine OpenAPI Spezifikation angewandt werden können. Metriken, welche mehr als einen Service untersuchen, können durch eine einzelne Spezifikation nicht untersucht werden, da eine Spezifikation nur einen Service beschreibt. Beispiele wären hier die Metrik *Number of Versions per Service*, wofür die totale Anzahl an Services benötigt wird und die Metrik *Service Interdependence im System*, bei der die Anzahl der Servicepaare untersucht wird, die bi-direktional voneinander abhängig sind. Zur groben Orientierung und weiteren Ideenfindung wurden vergleichbare Projekte gesucht und

¹<https://www.docker.com/>

deren Strukturen gesichtet. Die meisten relevanten Projekte haben dabei WSDL-Spezifikationen analysiert, da früher hauptsächlich Webservices basierend auf SOAP genutzt wurden und es kaum Spezifikationen zur Beschreibung von REST APIs gab. Es gab dennoch bereits Versuche, REST APIs auf REST Prinzipien und Richtlinien hin zu untersuchen, aber die Analysen wurden nicht anhand von maschinenlesbaren API Beschreibungen durchgeführt.

Erst seit 2011 etablierten sich mit Swagger und RAML erfolgreiche REST API Spezifikationen. Haupt et al. stellten in ihrem Projekt, welches in Kapitel 3.2 ausführlich beschrieben wird, ein Framework zur Analyse von RESTful APIs mithilfe der API Beschreibungssprachen vor. Dabei wurde auch der Sourcecode des Frameworks genauer untersucht und zur Ideensammlung genutzt. Die daraus resultierenden Erkenntnisse unterstützten den Entwurf eines Grundgerüsts für das OpenAPI Evaluationstool und dessen Umsetzung. Haupt et al. wandelten die API Beschreibungsformate in ein selbst definiertes, kanonisches Metamodell um, mit dessen Hilfe die Analyse durchgeführt wird. Dies wird für das System, das in dieser Bachelorarbeit entwickelt wird, nicht benötigt, da hier der Fokus lediglich auf der OpenAPI Spezifikation liegt. Das Projekt wird ausführlicher in dem Kapitel 3.2 beschrieben.

Nachdem das theoretische Wissen über die relevanten Techniken und Methoden aufgebaut wurde, ging es an die Überlegungen zur konkreten Umsetzung des Softwareprojekts. Hier lag zuerst die Wahl der Programmiersprache an. Schnell stellte sich heraus, dass für Java einige OpenAPI Parser Bibliotheken existieren und es viele Möglichkeiten gibt, REST Webservices für Java zu implementieren. Daher fiel die Entscheidung hier recht leicht.

Als Parser Bibliothek wurde der KaiZen-OpenApi-Parser² gewählt. Für diesen Parser existiert eine ausführliche und übersichtliche Dokumentation. Die offizielle Parser Bibliothek von Swagger kann zurzeit keine OpenAPI 3 Versionen bearbeiten und es ist keine Dokumentation vorhanden. Somit ist diese Möglichkeit direkt ausgeschlossen worden.

Zur Einbindung von externen Bibliotheken wurde Maven³ ausgewählt. Das ist ein Build Management Tool, mit dem auf einfache Art und Weise Bibliotheken eingebunden werden können, ohne diese in eine JAR-Datei exportieren zu müssen. Das hat den Vorteil, dass nicht nach jeder Änderung des importierten Projekts die JAR-Datei ersetzt werden muss.

Für den Wartungsbericht, der sowohl menschen- als auch maschinenlesbar sein soll, wurde das Ausgabeformat JSON gewählt. Dieses Format ist kompakt und durch die Textform auch für den Menschen lesbar. Jedes JSON-Dokument kann außerdem von JavaScript interpretiert werden. Als zweites Ausgabeformat wurde PDF gewählt, da es um einiges flexibler ist als JSON. Informationen können im PDF beliebig formatiert werden und es kann auch Multimedia beinhalten. Das ist besonders dann sinnvoll, wenn die ausgewerteten Ergebnisse zur Anschaulichkeit und Übersichtlichkeit in Diagrammen dargestellt werden sollen.

Mithilfe der genannten Werkzeuge wurde das Grundgerüst der Kernkomponente inklusive zwei Testmetriken programmiert. Die erste ausgewählte Metrik ist *Service Interface Data Cohesion* (SIDC), welche die Kohäsion des Systems beschreibt. Die Kohäsion wird berechnet, indem die Operationen paarweise im Hinblick auf die Ähnlichkeit der Parameter- und Rückgabedatentypen verglichen werden. Ein Service wird als kohäsiv angesehen, wenn alle möglichen Paare an Operationen mindestens einen gleichen Parameter- und Rückgabedatentyp besitzen. Hierbei wurde zusätzlich

²<https://github.com/RepreZen/KaiZen-OpenApi-Parser>

³<https://maven.apache.org/>

implementiert, dass atomare Datentypen wie String oder Integer nur dann berücksichtigt werden, wenn der Name des Parameters identisch ist. Diese Berücksichtigung ist wichtig, weil beispielsweise die Parameter „String customerId“ und „String petName“ nicht in Relation zueinander stehen, aber nach der Definition des Autors den Wert der Kohäsion erhöhen würden. Eine weitere Anpassung, die vom Autor nicht beschrieben wurde, ist die Gleichsetzung von Arrays von einem Datentyp mit einem einzelnen Vorkommen des Datentyps.

Die zweite Metrik *Weighted Service Interface Count* gibt die Anzahl der nach außen sichtbaren Operationen an und ist ein Maß für die Größe des Systems. Die Operationen sind basierend auf der Granularität sowie der Anzahl der Parameter unterschiedlich gewichtet. Der Autor der Metrik hat dabei keine Methoden definiert, wie die Gewichtung berechnet werden kann. Daher wird im Rahmen dieser Arbeit mit dem Standardgewicht 1 gearbeitet, was in der Zukunft angepasst werden kann.

Beide Messgrößen spielen bei Microservice-orientierten Systemen eine wichtige Rolle. Zudem können beide Metriken durch eine einzelne OpenAPI Spezifikation untersucht werden.

Nachdem das Grundgerüst der Kernkomponente stand, wechselte der Fokus zu den REST Web Services. In Java gibt es zum Beispiel die Bibliotheken Spring Boot⁴, Dropwizard⁵ oder Jersey⁶ zur Programmierung von REST Web Services. Die Wahl für dieses Projekt fiel auf Spring Boot, da es sich hierbei um eine einfache und verständliche Bibliothek mit vielen gut dokumentierten Anwendungsbeispielen handelt.

Um die Modularität zu gewährleisten, wurde ein separates Java-Projekt für die Webanwendung angelegt. Da kein Frontend existiert wurden die Funktionen der Webservices mit dem Programm Postman⁷ getestet. Postman ist eine API Entwicklungsumgebung und kann unter anderem HTTP-Anfragen an eine Webanwendung schicken, die normalerweise durch das Frontend gesendet werden.

Die Daten werden daraufhin in einer Datenbank gespeichert. Wichtig war, dass es sich um eine relationale Datenbank handelt, wobei hier jede passend war. Gewählt wurde schließlich MariaDB⁸, da es eine der erfolgreichsten relationalen Datenbanken ist.

Zum Schluss musste die Webanwendung in einen Docker-Container gebracht werden, sodass das Programm von jedem Gerät aus ohne Fehler ausgeführt werden kann. Hierzu wurden Dateien erstellt, sodass in der Dockerumgebung eine MariaDB-Datenbank und daraufhin die Webanwendung in einem Container gestartet werden. Dadurch lässt sich das Programm nun von jedem Gerät aus, auf dem Docker läuft, einfach bedienen.

Die leichte Erweiterbarkeit wurde im Rahmen der Bachelorarbeit nicht überprüft. Jedoch wurde der Code so implementiert, dass lediglich eine Metrik-Klasse programmiert werden soll, die von einem Interface erbt. Sobald eines der beiden Teilprogramme des Systems startet, werden automatisch alle implementierten Metriken für die Analyse verwendet. Für die leichte Erweiterbarkeit wäre es nun denkbar, dass ein weiterer Softwareentwickler eine Metrik-Klasse implementiert, wodurch der Aufwand für die Programmierung der Klasse untersucht werden kann.

⁴<https://github.com/spring-projects/spring-boot>

⁵<https://github.com/dropwizard/dropwizard>

⁶<https://jersey.github.io/>

⁷<https://www.getpostman.com/>

⁸<https://mariadb.org/>

4.3 Architektur

In der folgenden Abbildung ist eine grobgranulare Übersicht über das Gesamtprojekt dargestellt:

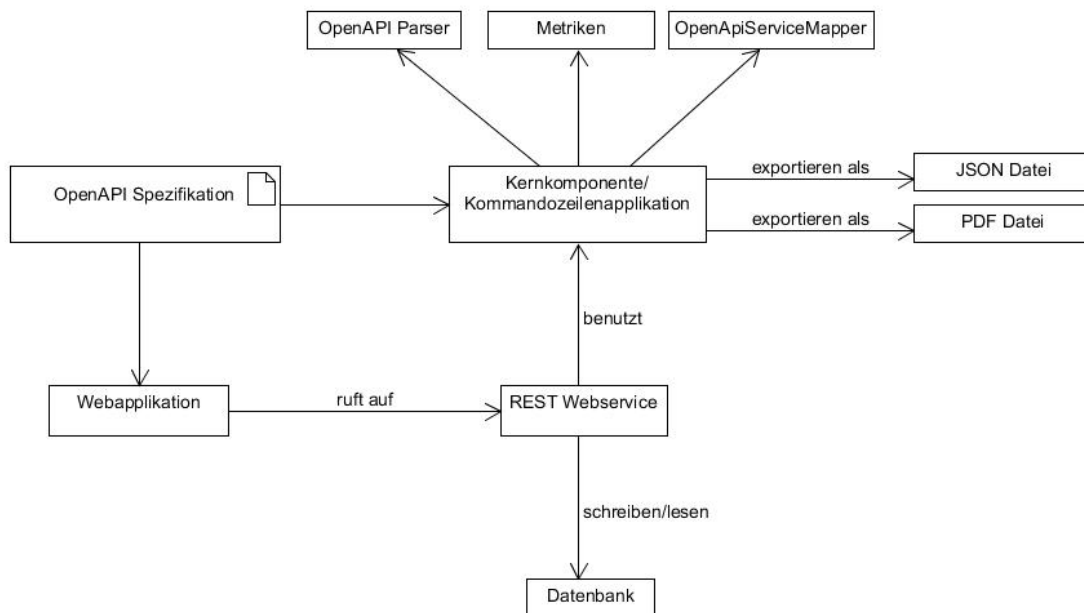


Abbildung 4.1: Übersicht über das Gesamtprojekt

Die OpenAPI Spezifikation kann entweder nur über die Kommandozeile oder über die Webapplikation eingelesen werden. Soll lediglich die Kommandozeilenapplikation benutzt werden, wird zuerst die OpenAPI Spezifikation über die Kommandozeile eingelesen. Die Applikation parsed die Spezifikation und wandelt sie in ein OpenApi3-Modell um. Dadurch können die Metriken mit den Objekten der Spezifikation arbeiten. Die Kommandozeilenapplikation ruft die verschiedenen Metriken-Klassen auf und lässt die Metriken berechnen. Daraufhin erstellt der OpenApiServiceMapper ein OpenApiService-Objekt, das allgemeine Informationen über die API und die Ergebnisse der einzelnen Metriken enthält. Dieses OpenApiService-Objekt kann nun entweder als JSON Datei oder als PDF Datei exportiert werden und bildet das Ergebnis des Prozesses.

Die Webapplikation baut auf der Kommandozeilenapplikation auf. Die OpenAPI Spezifikation wird über die Webapplikation eingelesen. Daraufhin wird ein REST Webservice gestartet, der die Klassen und Methoden der Kommandozeilenapplikation verwendet. Hier ist insbesondere der OpenAPI Parser der Kernkomponente für die Weitergabe des OpenAPI3 Modells an die Webapplikation und die Metrik-Klassen zur Berechnung der Metriken von Bedeutung. Der Webservice speichert diese Informationen in einer Datenbank, sodass die Ergebnisse der Evaluation jederzeit abgerufen werden können.

4.3.1 Aufbau des Gesamtsystems

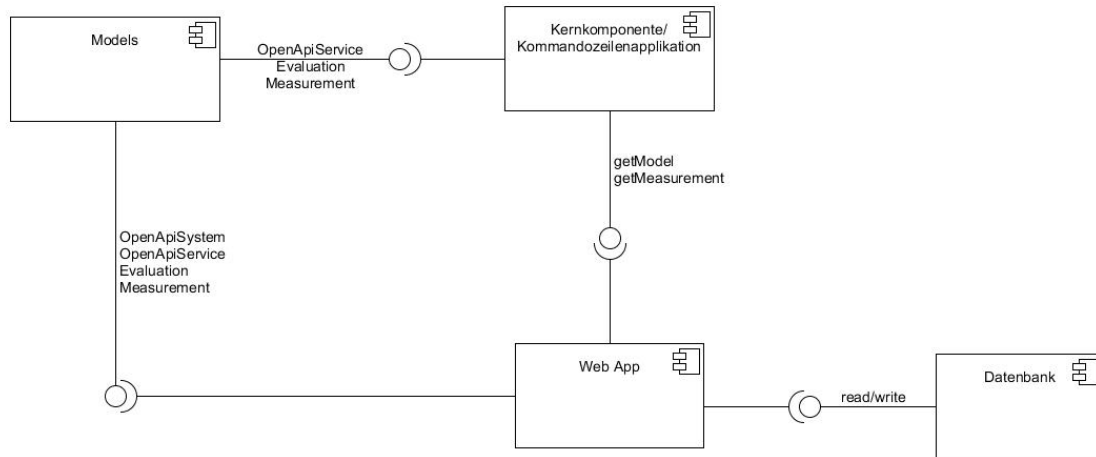


Abbildung 4.2: Komponentendiagramm

In der Abbildung 4.2 werden die Komponenten des Systems und ihre Beziehungen zueinander dargestellt.

Es gibt vier unterschiedliche Komponenten: die Modelle, die Kommandozeilenapplikation, die Webapplikation und die Datenbank. Die Komponente Modelle beinhaltet die Klassen `OpenApiSystem`, `OpenApiService`, `Evaluation` und `Measurement`. Diese Klassen sind für die Struktur der Ausgabe zuständig. Die Kommandozeilenapplikation und die Web-App bauen beide auf der Modell-Komponente auf, um ihre Ausgabe z.B. als JSON Datei oder als Antwort einer REST-Anfrage zu definieren.

In der Kommandozeilenapplikation, also der Kernkomponente, werden Objekte aus den Klassen `OpenApiService`, `Evaluation` und `Measurement` erstellt. In `OpenApiService` werden Informationen über den Titel der API und den Speicherpfad, also den Ort an der die API lokalisiert ist, gespeichert. Dieser Speicherpfad kann dabei sowohl im Web sein, als auch lokal auf dem Computer. Außerdem kann der `OpenApiService` eine Liste an Evaluationen speichern, wobei in der Kernkomponente nur eine einzige Evaluation durchgeführt wird, die alle anwendbaren Metriken enthält.

Die `Evaluation` enthält Informationen über die API Version, den Namen der OpenAPI Spezifikation, die Swagger Version und das Datum, an dem die Evaluation durchgeführt wurde. Evaluationen besitzen zusätzlich noch eine Liste an `Measurements`. Da in der Kernkomponente immer sämtliche Metriken analysiert werden, ist die Anzahl der `Measurement`-Objekte von der Anzahl der implementierten Metriken abhängig.

In der Klasse `Measurement` werden der Name der Metrik, der errechnete Wert der Metrik und weitere frei wählbare Zusatzinformationen definiert.

Aus dem `OpenApiService` kann nun durch den `OpenApiServiceMapper` aus Abbildung 4.1 ein JSON-Objekt für den JSON-Export oder eine `HashMap` für den PDF-Export gebaut werden.

Die Webapplikation benutzt von der Modell-Komponente die Klassen `OpenApiSystem`, `OpenApiService`, `Evaluation` und `Measurement`. Ein `OpenApiSystem` ist lediglich ein Überordner, um eine Menge an `OpenApiServices` zu speichern. Dementsprechend besteht es nur aus einem Systemnamen

und einer Liste von OpenApiServices. Diese Struktur ist sinnvoll, wenn unterschiedliche REST APIs miteinander verglichen werden sollen, da pro OpenApiService nur eine einzige Spezifikation und damit auch nur eine einzige API untersucht werden kann. Mit einem Vergleich von verschiedenen APIs können unter anderem Durchschnitts- und Ausreißerwerte für bestimmte Metriken gefunden werden, womit dann analysiert werden kann, wie die Struktur der analysierten API im Vergleich zu anderen APIs ist.

Im OpenApiService der Web-App können im Vergleich zur Kernkomponente mehrere Evaluationen enthalten sein. Die Besonderheit hierbei ist, dass der Nutzer auswählen kann, welche Metriken er untersuchen will und nicht alle vorhandenen Metriken analysiert werden müssen. Will der Nutzer in einer Evaluation im Nachhinein weitere Metriken untersuchen, können in einer Anfrage zusätzliche Measurements definiert werden, die daraufhin analysiert und in der Evaluation ergänzt werden.

Die Webapplikation benutzt die Kernsystem-Komponente, um die eingelesene Spezifikation zu parsen und ein OpenApi3 Modell zu holen. Je nachdem, welche Metriken in der Anfrage beim Erstellen der Evaluation ausgewählt wurden, werden diese berechnet und die Ergebnisse als Measurement-Objekte von der Kernkomponente geholt. Diese Menge an Measurements wird dann der zugehörigen Evaluation hinzugefügt.

Die erstellten Objekte werden in der Datenbank gespeichert und können in der Zukunft aufgerufen und gelöscht werden.

Für die Kommunikation mit den REST Webservices wurde das Tool Postman benutzt. Damit ist es möglich, HTTP Anfragen an einen Webservice zu schicken. Soll nun in der Webapplikation ein OpenApiSystem erstellt werden, muss eine POST-Anfrage an der richtigen URI im definierten JSON-Format an den Webservice geschickt werden.

Die POST-Anfrage zur Erstellung eines OpenApiSystems ist in */api/systems* und würde folgendermaßen aussehen:

```
{
  "name" : "testSystem1"
}
```

Daraufhin wird ein OpenApiSystem mit dem Namen „testSystem1“ und einer eindeutigen id erstellt. Diese id wird in der Datenbank als Primärschlüssel gespeichert. Um ein OpenApiSystem wieder aufzurufen, kann nun eine GET-Anfrage geschickt werden. Dafür wird ein Parameter, in diesem Fall die id zur eindeutigen Identifikation der Ressource, zur URI hinzugefügt.

Im oben genannten Fall würde die URI so aussehen: */api/system/1*

Wird zu der gleichen URI nun eine DELETE-Anfrage geschickt, wird die entsprechende Ressource gelöscht. Wird bei einer GET-Anfrage oder DELETE-Anfrage auf eine nicht existierende Ressource zugegriffen, so kommt eine *ResourceNotFoundException*.

Bei den Klassen OpenApiService, Evaluation und Measurements funktionieren die HTTP-Anfragen ähnlich. Diese HTTP-Anfragen werden in Kapitel 4.4 durch ein Anwendungsbeispiel genauer beschrieben.

4.3.2 Aufbau der Teilsysteme

In diesem Abschnitt wird ein kurzer Überblick über die Struktur der beiden Teilsysteme gegeben. Die Abbildungen 4.3 und 4.4 zeigen dabei die Paketdiagramme der beiden Systeme.

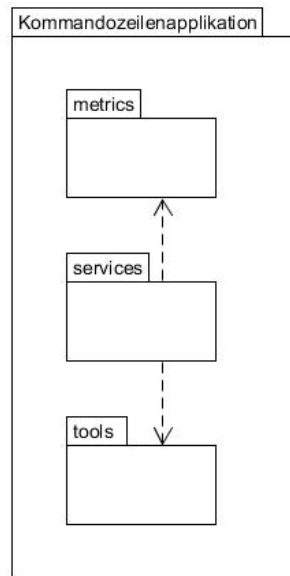


Abbildung 4.3: Paketdiagramm Kommandozeilenprogramm

Die Kommandozeilenapplikation besteht aus drei Paketen: *services*, *metrics* und *tools*. Das Paket *services* enthält die gesamte Funktionalität des Programms. Hier wird die Logik ausgeführt und die Dateien exportiert. Das Paket *tools* enthält die Werkzeuge für das Parsen, das Transformieren in JSON oder PDF und die Ausgabe der Dateien, welche von *services* benutzt werden. Außerdem führt *services* die Logik von allen Metriken im Paket *metrics* aus und holt sich die Informationen, um damit weiterzuarbeiten.

Die Einlesung der OpenAPI Spezifikation erfolgt durch Argumente, welche beim Start des Programms eingegeben werden. Außerdem wird in den Argumenten auch ausgewählt, welches Ausgabeformat die Datei haben soll und wo die Ausgabe gespeichert wird. Das Parsen der Argumente wurde mit Hilfe der Apache Commons CLI Bibliothek⁹ realisiert. Folgende Argumente können benutzt werden:

- `-uri http://url-to-swagger-file.com`
Das Argument `-uri` wird benutzt, wenn eine OpenAPI Spezifikationsdatei im Internet zu finden ist.
- `-file path/to/file.yaml` oder `-file path/to/file.json`
Das Argument `-file` wird für lokale Spezifikationen benutzt.
- `-pdf path/to/outputfile.pdf`
Das Argument `-pdf` wird benutzt, falls die Ausgabedatei im PDF Format sein soll. Es muss ein Pfad spezifiziert werden, hinter dem die Datei gespeichert werden soll.

⁹<https://github.com/apache/commons-cli>

- `-json path/to/outputfile.json`

Das Argument `-json` wird benutzt, falls die Ausgabedatei im JSON Format sein soll. Es muss ein Pfad spezifiziert werden, hinter dem die Datei gespeichert werden soll.

Zu beachten ist, dass eines der beiden Argumente `-uri` oder `-file` immer vorhanden sein muss. Sowohl `-pdf` als auch `-json` sind dagegen optional.

Für die JSON-Datei wurde die Bibliothek GSON¹⁰ benutzt. Diese ist zuständig für die Erstellung des maschinenlesbaren JSON-Objekts und die Formatierung für die bessere Menschenlesbarkeit. Die Exportfunktion von PDF-Dateien wird über die Bibliothek itextpdf¹¹ realisiert. Um diese Bibliothek zu benutzen muss zuvor eine HashMap der Ausgabe erstellt werden. Diese HashMap wird daraufhin von itextpdf in ein übersichtliches Format umgewandelt und als PDF ausgegeben.

Für die Metriken wurde das Interface *IMetric* angelegt, von der sämtliche Metrik-Klassen erben. Wenn eine neue Metrik implementiert werden soll, sollten die Methoden des Interface darin enthalten sein. Mithilfe der Java Reflections Bibliothek wird beim Start des Programms nach sämtlichen Klassen gescannt, die vom IMetric-Interface erben. Die gefundenen Metrik-Klassen werden dann für die Analyse der Spezifikation verwendet. Somit ist auch eine gute Erweiterbarkeit des Gesamtprojekts gewährleistet, da für das Anlegen einer neuen Metrik nur die Metrik-Klasse implementiert werden muss. Die Klasse wird durch das dynamische Laden mithilfe der Java-Reflections Bibliothek automatisch in das Programm eingebunden.

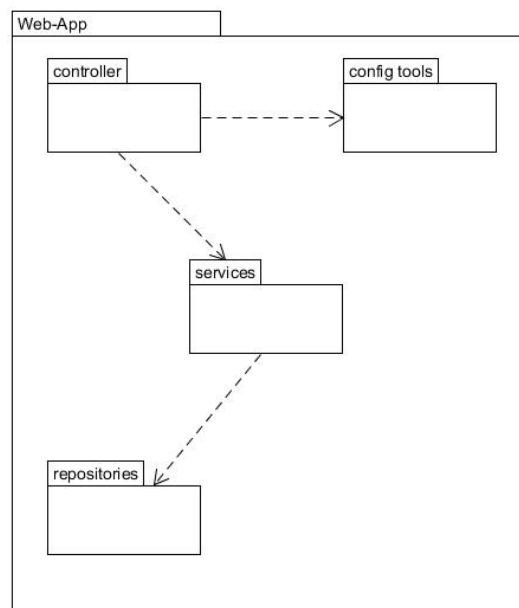


Abbildung 4.4: Paketdiagramm Webapplikation

¹⁰<https://github.com/google/gson>

¹¹<https://github.com/itext/itextpdf>

Die Webanwendung umfasst die Pakete *controller*, *services*, *repositories* und *config tools*. Die Abhängigkeiten der Pakete zueinander werden in Abbildung 4.4 dargestellt.

Dabei steht in *services* die Logik der Anwendung. Hier wird auf die Kernkomponente zugegriffen, um die Funktionalität der HTTP-Methoden zu implementieren. Der Controller greift auf diese HTTP-Methoden zu und bindet sie an HTTP-Anfragen. So werden beispielsweise alle Anfragen für das OpenApiSystem an „api/systems“ gemappt. Um das System mit der id=1 zu bekommen, muss eine GET-Anfrage in „api/systems/1“ gestellt werden. In *config tools* wird definiert, wie die HTTP-Anfrage aussehen soll. Zum Beispiel muss beim Erstellen eines OpenApiSystems ein Name in der POST-Anfrage mitgegeben werden, damit das System unter dem Namen gespeichert werden kann.

Der REST Webservice wurde mit dem Spring Boot Framework implementiert. Spring Boot erlaubt es, selbstständig laufende Java-Applikationen zu bauen, die über `java -jar` gestartet werden können. Außerdem hat Spring Boot mit Tomcat einen Webserver direkt eingebettet. Es können aber auch die Webserver-Alternativen Jetty oder Undertow gewählt werden.

Die erstellten Objekte werden daraufhin in einem `CrudRepository`¹² gespeichert. Das `CrudRepository` wurde im Paket *repositories* erstellt und bietet die typischen CRUD-Funktionalitäten an. Diese sind [Sdr18]:

- Entität speichern. Hier kann auch gleichzeitig eine Menge an Entitäten gespeichert werden.
- Entität finden und zurückgeben.
- Alle Entitäten zurückgeben.
- Die Anzahl an Entitäten in der Tabelle zurückgeben.
- Entität finden und löschen.
- Verifizieren, ob eine Entität existiert, durch Übergabe des Primärschlüssels.

Um die Einträge im Repository persistent zu machen, wurde die persistence-Bibliothek¹³ benutzt. Diese Bibliothek ist eine Implementierung der Spezifikation Java Persistence API (JPA). Über JPA kann der Entwickler Daten aus relationalen Datenbanken auf Java-Objekte abbilden, speichern, updaten sowie abrufen und umgekehrt. [Jpa18]

Als relationale Datenbank wurde MariaDB gewählt. Die Struktur der Datenbank des Evaluationstools wird in Abbildung 4.5 dargestellt.

¹²<https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html>

¹³<https://github.com/eclipse/javax.persistence>

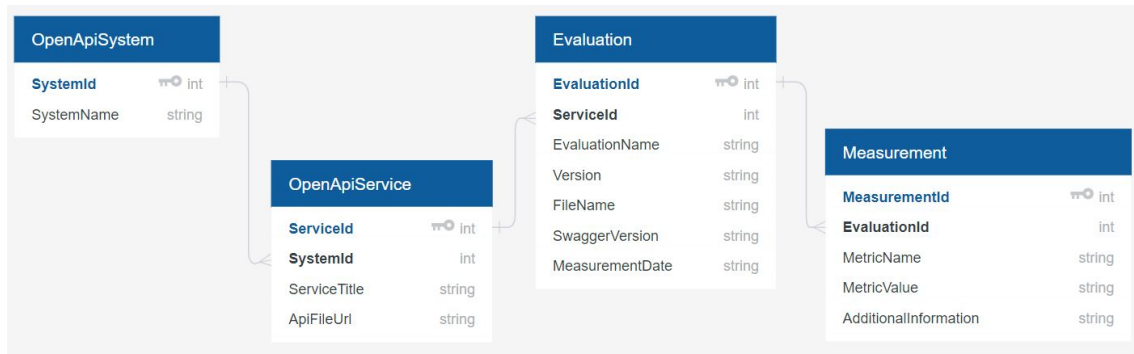


Abbildung 4.5: Datenbankdiagramm

4.4 Anwendungsbeispiel

Im folgenden Abschnitt werden mögliche Anwendungsfälle der Kommandozeilen- und der Webapplikation sowie deren Ablauf beschrieben. In den Abbildungen 4.6 und 4.7 werden über Use Case-Diagramme Übersichten über sämtliche Anwendungsfälle der beiden Systeme gegeben. Daraufhin wird in Anwendungsbeispielen gezeigt, wie diese Anwendungsfälle umgesetzt werden können.

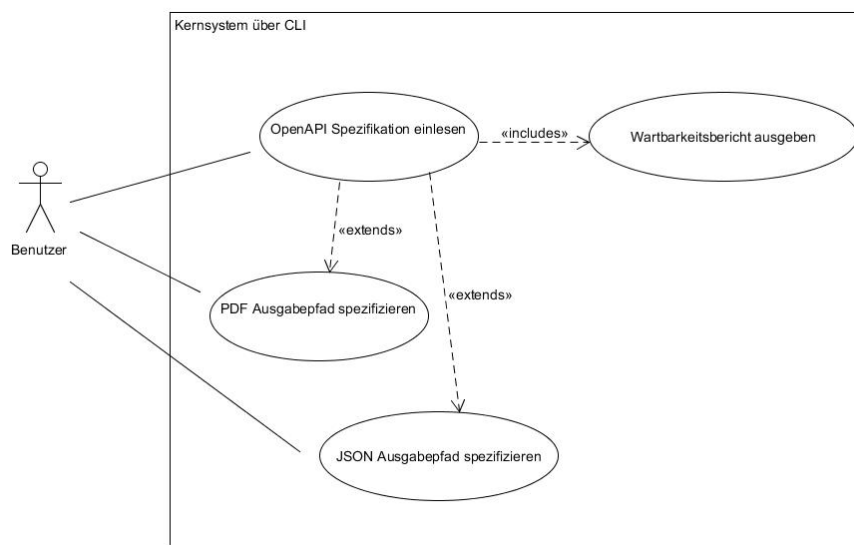


Abbildung 4.6: Use Case-Diagramm Kommandozeilenapplikation

4.4.1 Anwendungsbeispiel Kommandozeilenapplikation

Um über die Kommandozeilenapplikation eine OpenAPI Spezifikation einzulesen, muss das Programm mit Argumenten gestartet werden. Die möglichen Argumente wurden in Kapitel 4.3.2 bereits genauer beschrieben. Angenommen, die ausführbare JAR-Datei ist in `./target` gespeichert und hat den

4 Entwicklung eines OpenAPI Evaluationstools

Namen `openapi-evaluation-cli`. Soll nun zum Beispiel die Swagger Petstore Beispielspezifikation von Github eingelesen werden und sowohl in PDF als auch in JSON in `C:\Ausgabe` gespeichert werden, so wird folgender Befehl in die Kommandozeile eingegeben:

```
java -jar ./target/openapi-evaluation-cli.jar -uri https://raw.githubusercontent.com/OAI/
OpenAPI-Specification/master/examples/v3.0/petstore-expanded.yaml -pdf C:\Ausgabe\
Wartbarkeitsbericht.pdf -json C:\Ausgabe\Wartbarkeitsbericht.json
```

Daraufhin wird in der Konsole der Wartbarkeitsbericht zurückgegeben und zusätzlich in `C:\Ausgabe` als `Wartbarkeitsbericht.pdf` und `Wartbarkeitsbericht.json` gespeichert. Die Ausgabe der Konsole und der Bericht in JSON sieht für dieses Beispiel folgendermaßen aus:

```
{
  "apiTitle": "Swagger Petstore",
  "apiFileUrl": "https://raw.githubusercontent.com/OAI/OpenAPI-Specification/master/examples/
v3.0/petstore-expanded.yaml",
  "apiVersion": "1.0.0",
  "apiFileName": "petstore-expanded.yaml",
  "swaggerVersion": "3.0.0",
  "measurementDate": "2018/09/11",
  "measurement": [
    {
      "metricName": "WeightedServiceInterfaceCount",
      "metricValue": 4.0,
      "additionalInformation": "Number of different operations: 4"
    },
    {
      "metricName": "ServiceInterfaceDataCohesion",
      "metricValue": 0.58,
      "additionalInformation": "Set of pairwise operations with at least one common parameter:
[[find pet by id, deletePet]]Set of pairwise operations with common return type: [[
deletePet, addPet], [find pet by id, deletePet], [find pet by id, addPet], [find pet by
id, findPets], [deletePet, findPets], [addPet, findPets]]Number of operations: 4.0"
    }
  ]
}
```

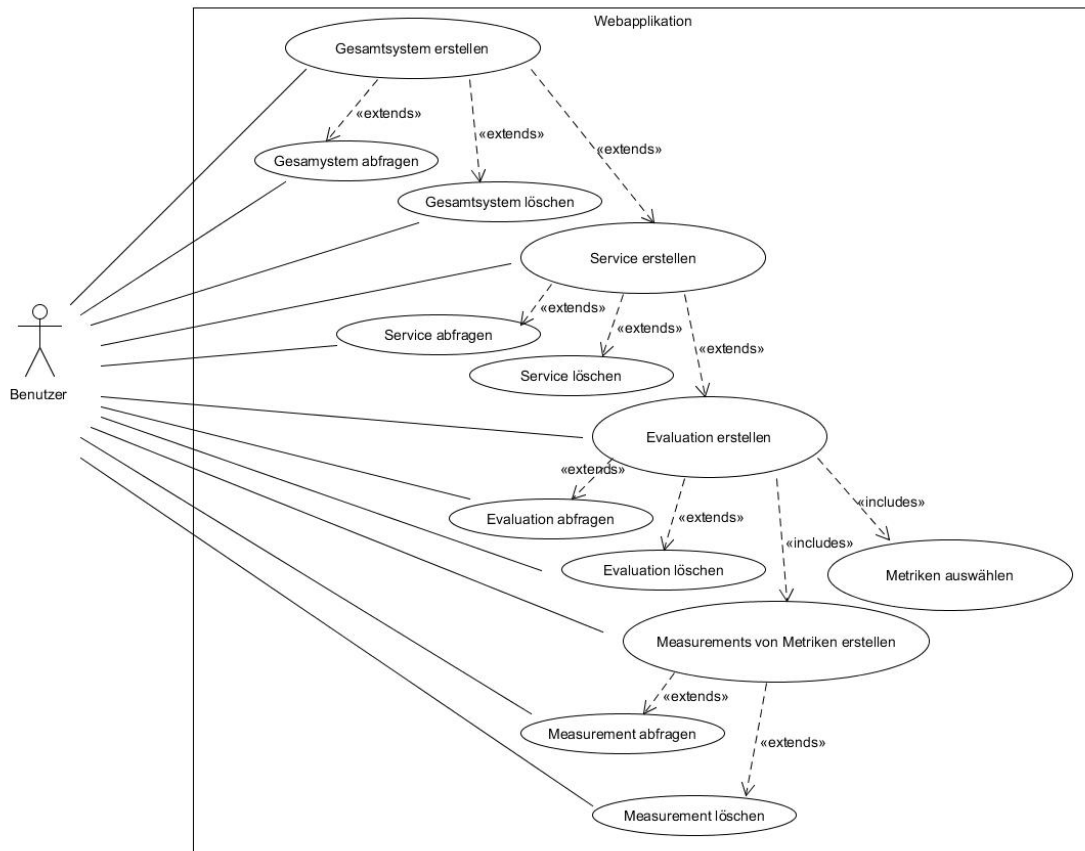


Abbildung 4.7: Use Case-Diagramm Webapplikation

4.4.2 Anwendungsbeispiel Webapplikation

Bei der Webapplikation kann das Programm wieder als JAR-Datei gestartet werden. Alternativ ist auch das Starten in einem Docker-Container möglich. Zum Starten als JAR-Datei wird der *java-jar* Befehl in der Kommandozeile verwendet. Hierfür werden keine Argumente benötigt, weil die Einlesung der Spezifikation bei der Webanwendung über eine HTTP-Anfrage funktioniert. Zum Starten der Applikation in einem Docker-Container muss zuerst Docker gestartet werden. Daraufhin wird in dem Ordner, in dem die zwei Dateien „Dockerfile“ und „docker-compose.yml“ des Projektes lokalisiert sind, der Befehl *docker-compose up* in die Kommandozeile eingegeben. Daraufhin startet im Docker-Container die MariaDB-Datenbank und die Webapplikation. Der Webserver wird dabei auf dem Port 8080 gestartet, sodass die Webanwendung über *localhost:8080* aufgerufen werden kann.

Um mit der Webanwendung zu kommunizieren, kann das Programm Postman benutzt werden. Um ein OpenApiSystem mit dem Namen „testSystem1“ zu erstellen, wird eine POST-Anfrage in *api/systems* gestellt. Der Content-Typ hierbei muss auf *application/json* gesetzt werden. Die Anfrage lautet:

```
{
  "name": "testSystem1"
}
```

Als Antwort wird folgendes zurückgegeben:

```
{
  "id": 1,
  "systemName": "testSystem1",
  "openApiServiceList": []
}
```

Nachdem das System auf der id=1 erstellt wurde, ist dieses System auf die URL *api/systems/1* gemappt. Mithilfe einer GET-Anfrage oder einer DELETE-Anfrage auf dieser URL kann das System aufgerufen oder gelöscht werden. Soll nun in testSystem1 eine Klasse OpenApiService erstellt werden, so muss auf *api/systems/1/services* eine POST-Anfrage gestellt werden. Bei der Anfrage muss ein Pfad zu der OpenAPI Spezifikation angegeben werden. Angenommen, es soll wie oben der Swagger Petstore analysiert werden, so sieht die Anfrage folgendermaßen aus:

```
{
  "apiFileUrl": "https://raw.githubusercontent.com/OAI/OpenAPI-Specification/master/examples/v3.0/petstore-expanded.yaml"
}
```

Daraufhin wird ein OpenApiService mit der id=1 erstellt. Als Antwort der POST-Anfrage wird das gesamte OpenApiSystem zurückgegeben. Der OpenApiService ist dabei in openApiServiceList gespeichert. Mithilfe einer GET-Methode oder DELETE-Methode in *api/systems/1/services/1* kann der OpenApiService zurückgegeben oder gelöscht werden. Der erstellte OpenApiService sieht wie folgt aus:

```
{
  "id": 1,
  "serviceTitle": "Swagger Petstore",
  "apiFileUrl": "https://raw.githubusercontent.com/OAI/OpenAPI-Specification/master/examples/v3.0/petstore-expanded.yaml",
  "evaluationList": []
}
```

Entsprechend wird auch eine Evaluation erstellt. Hierfür muss in *api/systems/1/services/1/evaluation* eine POST-Anfrage mit dem Namen der Evaluation und einer Liste von Metriken gestellt werden. Die Metriken müssen genau so eingetragen werden, wie die Namen der Metriken definiert sind. Eine Anfrage mit dem Namen „testEvaluation1“ und den Metriken ServiceInterfaceDataCohesion und WeightedServiceInterfaceCount hat die folgende Struktur:

```
{
  "evaluationName": "testEvaluation1",
  "metricName" : ["ServiceInterfaceDataCohesion", "WeightedServiceInterfaceCount"]
}
```


Durch diese Anfrage wird eine Evaluation mit der `id=1`, dem Namen „testEvaluation1“ und einer Liste an Measurements erstellt. Die Measurements enthalten dabei den Namen, das Ergebnis der Metrik und noch flexible zusätzliche Informationen über die Metrik. In der Evaluation sind außerdem noch zusätzliche Informationen über die analysierte OpenAPI Spezifikation vorhanden. Durch eine GET-Methode in `api/systems/1/services/1/evaluation/1` kann die Evaluation dann zurückgegeben werden. Diese sieht wie folgt aus:

```
{
  "id": 1,
  "evaluationName": "evaluationName1",
  "version": "1.0.0",
  "fileName": "petstore-expanded.yaml",
  "swaggerVersion": "3.0.0",
  "measurementDate": "2018/10/08",
  "measurementList": [
    {
      "id": 1,
      "metricName": "ServiceInterfaceDataCohesion",
      "metricValue": 0.58,
      "additionalInformation": "Set of pairwise operations with at least one
common parameter:[[find pet by id, deletePet]]Set of pairwise operations
with common return type: [[deletePet, addPet], [find pet by id, deletePet
], [find pet by id, addPet], [find pet by id, findPets], [deletePet,
findPets], [addPet, findPets]]Number of operations: 4.0"
    },
    {
      "id": 2,
      "metricName": "WeightedServiceInterfaceCount",
      "metricValue": 4,
      "additionalInformation": "Number of different operations: 4"
    }
  ]
}
```

Sollen dieser `measurementList` weitere Measurements hinzugefügt werden, kann eine POST-Anfrage in `api/systems/1/services/1/evaluation/1` mit dem Namen der Metrik gestellt werden:

```
{
  "metricName" : "ServiceInterfaceDataCohesion"
}
```

Die Measurements können über ihre `id` mit einer GET-Anfrage zurückgegeben oder mit einer DELETE-Anfrage gelöscht werden.

5 Fazit

In dieser Arbeit wurde ein prototypisches System zur automatischen Analyse von REST APIs mithilfe der OpenAPI Spezifikation entwickelt. Dadurch wurde es ermöglicht, die Wartbarkeit eines Services automatisch zu analysieren. Die Wartbarkeit ist ein wichtiges Qualitätsattribut von Software und kann sehr gut anhand der Struktur eines Systems analysiert werden. Zur Bewertung der Wartbarkeit wurden beispielhaft zwei Metriken benutzt: *Service Interface Data Cohesion* (SIDC) und *Weighted Service Interface Count* (WSIC). SIDC ist eine Metrik, die sich auf die Kohäsion der Operationen eines Dienstes bezieht. Die Kohäsion wird berechnet, indem die Operationen paarweise im Hinblick auf die Ähnlichkeit der Parameter- und Rückgabedatentypen verglichen werden. WSIC ist eine Metrik zur Untersuchung der Größe des Systems. Sie beschreibt die Menge aller offenliegenden Schnittstellenoperationen eines Dienstes. Dabei haben die Operationen basierend auf der Anzahl der Parameter oder der Granularität unterschiedliche Gewichte, die für die Berechnung der Metrik berücksichtigt werden sollen.

Das entworfene System besteht aus zwei eigenständigen Komponenten: einer Kernkomponente, welche über die Kommandozeile gestartet werden kann und einer Webapplikation, die auf der Kernkomponente aufbaut und die Funktionalität des Systems über RESTful Webservices anbietet. Bei der Kernkomponente wird die OpenAPI Spezifikation über die Kommandozeile eingelesen, geparsed und in ein OpenAPI3-Modell umgewandelt. Dann werden die Metriken angewandt, ein `OpenApiService`-Objekt wird erstellt, das die Ergebnisse der Analyse enthält und ein Wartbarkeitsbericht wird ausgegeben.

Bei der Webapplikation wird die OpenAPI Spezifikation über eine HTTP-Anfrage eingelesen, daraufhin startet der RESTful Webservice, der die Klassen und Methoden der Kernkomponente verwendet, um die Funktionalität des Kernsystems über HTTP-Methoden anzubieten. Die Ergebnisse der Analyse werden anschließend in einer Datenbank gespeichert.

Der Wartbarkeitsbericht wird in den Formaten JSON und PDF angeboten und ist somit sowohl menschen- als auch maschinenlesbar. Die gesamte Software ist dabei modular und kann dem Wartbarkeitsgedanken folgend einfach erweitert werden. Zur besseren Portabilität ist ein Docker-Container vorhanden, sodass das System auf allen Plattformen, auf denen Docker laufen kann, ohne Probleme ausgeführt werden kann.

5.1 Limitationen der Arbeit

Trotz aller Auseinandersetzung mit den relevanten Themen und Überlegungen zur Umsetzung, ist dieses Projekt, wie jedes andere auch, mit Einschränkungen versehen.

Ein Kritikpunkt bezieht sich auf die angewandten Metriken. Die Metrik WSIC kann kritisch betrachtet werden, diese beinhaltet in ihrer Berechnung gewichtete Operationen. Allerdings wurde in den wissenschaftlichen Arbeiten hierzu nicht genau beschrieben, wie die Operationen zu gewichten sind, daher wurde in dieser Arbeit mit dem Standardwert 1 für jede Operation gerechnet. Selbstverständlich kann dieser Wert angepasst werden.

Die Metrik SIDC zur Beurteilung der Kohäsion, wurde in dieser Art nur von Perepletchikov et al. [PRF07] aufgestellt. Zwar wurde ihre Bedeutung begründet, jedoch ist die Kohäsion im Allgemeinen nicht einfach zu analysieren, da sie zum großen Teil semantischer Natur ist. Generell sollten Metriken kritisch betrachtet werden. Ihre Aussagekraft und Passgenauigkeit zum jeweiligen Problem muss analysiert werden und eine überlegte Auswahl getroffen werden, um aussagekräftige Ergebnisse erzielen zu können.

Auch können nicht alle Teile der Wartbarkeit anhand der OpenAPI Spezifikation analysiert werden, wie zum Beispiel die von Senivongse und Puapolthep in Kapitel 2.4.2 definierten low-level Qualitätsattribute Erreichbarkeit und Verfügbarkeit. Die Erreichbarkeit besagt, wie einfach der Service während der Entwicklungs- und Wartungsphase erreicht werden kann. Die Verfügbarkeit bezieht sich auf die Häufigkeit und Länge der Ausfallzeit, wenn der Service Wartungen und Änderungen am System durchführt. Dies sind keine Attribute, die anhand einer strukturellen Analyse bestimmt werden können.

Neben den Metriken ist auch die Möglichkeit zum Vergleichen von mehreren Schnittstellenspezifikationen ausbaubar. Momentan werden die Metriken auf einzeln eingelesene Dokumente angewandt und die Ergebnisse in einer Datenbank gespeichert. Somit kann pro Analyse nur ein einzelner Service untersucht werden. Die gespeicherten Dokumente können dann miteinander verglichen werden. Eine Analyse mehrerer Spezifikationen gleichzeitig ist momentan nicht möglich, kann aber in der Zukunft leicht eingebaut werden.

5.2 Ausblick

Da das System durch das bestehende Wissen über Wartbarkeit und damit zusammenhängend auch Erweiterbarkeit auch so gestaltet wurde, können fehlende Teile leicht ergänzt werden. Neben dem hier betrachteten Aspekt der Wartbarkeit gibt es noch weitere Attribute, die eine Aussage über die Qualität der RESTful API tätigen können. Diese können zur umfassenderen Analyse der Schnittstelle dem System ergänzt werden.

Eine weitere sinnvolle Erweiterung ist eine grafische Benutzeroberfläche. Eine ansprechende und intuitive Oberfläche führt zu einer besseren Nutzererfahrung. Außerdem können die HTTP-Methoden des Webservices ohne Zusatzwerkzeuge wie zum Beispiel Postman benutzt werden.

Die Ergebnisse der Metriken der verschiedenen Evaluationen können in der Zukunft auch grafisch veranschaulicht werden, z.B. in Form von Diagrammen. So können Unregelmäßigkeiten und Durchschnitte besser und schneller erkannt werden.

Es wurde in diesem Projekt nur die Analyse einzelner Services umgesetzt, in der Zukunft könnte eine API-weite Analyse durch gleichzeitiges Einlesen von mehreren bzw. allen Spezifikationen eines Systems implementiert werden. Somit können weitere Metriken implementiert werden, die

zurzeit nicht angewendet werden können. Ein Beispiel wäre hier die Metrik *Service Interdependence in the System*, bei der die Anzahl der Servicepaare gesucht wird, die bi-direktional voneinander abhängig sind.

Literaturverzeichnis

- [BBMP08] C. Bartolini, A. Bertolino, E. Marchetti, A. Polini. „Towards automated WSDL-based testing of web services“. In: *International Conference on Service-Oriented Computing*. Springer. 2008, S. 524–529 (zitiert auf S. 37).
- [BCSV12] R. Baggen, J. P. Correia, K. Schill, J. Visser. „Standardized code quality benchmarking for improving software maintainability“. In: *Software Quality Journal* 20.2 (2012), S. 287–307 (zitiert auf S. 32).
- [BM11] D. Baski, S. Misra. „Metrics suite for maintainability of extensible markup language Web Services“. In: *IET software* 5.3 (2011), S. 320–341 (zitiert auf S. 37).
- [BWZ17] J. Bogner, S. Wagner, A. Zimmermann. „Automatically measuring the maintainability of service- and microservice-based systems: a literature review“. In: *Proceedings of the 27th International Workshop on Software Measurement and 12th International Conference on Software Process and Product Measurement*. ACM. 2017, S. 107–115 (zitiert auf S. 34).
- [CDK+02] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, S. Weerawarana. „Unraveling the Web services web: an introduction to SOAP, WSDL, and UDDI“. In: *IEEE Internet computing* 6.2 (2002), S. 86–93 (zitiert auf S. 17).
- [CH09] J.-C. Chen, S.-J. Huang. „An empirical analysis of the impact of software development problem factors on software maintainability“. In: *Journal of Systems and Software* 82.6 (2009), S. 981–992 (zitiert auf S. 30).
- [Dai11] R. Daigneau. *Service Design Patterns: fundamental design solutions for SOAP/WSDL and restful Web Services*. Addison-Wesley, 2011 (zitiert auf S. 12, 16).
- [DFoCF18] *Design First or Code First: What’s the Best Approach to API Development?* <https://swagger.io/blog/api-design/design-first-or-code-first-api-development/>. Accessed: 2018-10-01 (zitiert auf S. 25–27).
- [EGN+14] T. Erl, C. Gee, H. Normann, J. Kress, B. Maier. *Next generation SOA: A concise introduction to service technology & service-orientation*. Pearson Education, 2014 (zitiert auf S. 12, 13).
- [EntOAS18] *Einführung in Swagger: Mehr als nur Schnittstellenbeschreibung*. <https://entwickler.de/online/web/openapi-swagger-579827368.html>. Accessed: 2018-10-01 (zitiert auf S. 28).
- [Evo18] *Is Design Dead?* <https://martinfowler.com/articles/designDead.html>. Accessed: 2018-09-21 (zitiert auf S. 15).
- [FBB+99] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999 (zitiert auf S. 30, 38).

- [FBT15] C. Fadel, M. Bialik, B. Trilling. *Four-dimensional education: The competencies learners need to succeed*. Center for Curriculum Redesign, 2015 (zitiert auf S. 21).
- [FGM+99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee. *Hypertext transfer protocol–HTTP/1.1*. Techn. Ber. 1999 (zitiert auf S. 16).
- [FL15] M. Fowler, J. Lewis. „Microservices: Nur ein weiteres Konzept in der Softwarearchitektur oder mehr“. In: *Objektspektrum* 1.2015 (2015), S. 14–20 (zitiert auf S. 15).
- [Fow10] M. Fowler. „Richardson Maturity Model: steps toward the glory of REST“. In: *Online at <http://martinfowler.com/articles/richardsonMaturityModel.html>* (2010), S. 24–65 (zitiert auf S. 20).
- [FT00] R. T. Fielding, R. N. Taylor. *Architectural styles and the design of network-based software architectures*. Bd. 7. University of California, Irvine Doctoral dissertation, 2000 (zitiert auf S. 18).
- [Gal03] D. Galin. „Software quality metrics—from theory to implementation“. In: *Software Quality Professional* 5.3 (2003), S. 24 (zitiert auf S. 30).
- [GK91] G. K. Gill, C. F. Kemerer. „Cyclomatic complexity density and software maintenance productivity“. In: *IEEE transactions on software engineering* 17.12 (1991), S. 1284–1288 (zitiert auf S. 30).
- [Has96] K. Hashim. „A software maintainability attributes model“. In: *Malaysian Journal of Computer Science* 9.2 (1996), S. 92–97 (zitiert auf S. 31).
- [HLV18] F. Haupt, F. Leymann, K. Vukojevic-Haupt. „Api governance support through the structural analysis of rest apis“. In: *Computer Science-Research and Development* 33.3-4 (2018), S. 291–303 (zitiert auf S. 40).
- [Jax14] *Wer REST will, muss mit HATEOAS ernst machen*. <https://jaxenter.de/wer-rest-will-muss-mit-hateoas-ernst-machen-489>. Accessed: 2018-09-27 (zitiert auf S. 20).
- [JK99] J. J. Jiang, G. Klein. „Risks to different aspects of system success“. In: *Information & Management* 36.5 (1999), S. 263–272 (zitiert auf S. 30).
- [Jon04] C. Jones. „Software project management practices: Failure versus success“. In: *CrossTalk: The Journal of Defense Software Engineering* 17.10 (2004), S. 5–9 (zitiert auf S. 30).
- [Jpa18] *Java Persistence API*. <https://www.oracle.com/technetwork/java/javasee/tech/persistence-jsp-140049.html>. Accessed: 2018-10-04 (zitiert auf S. 52).
- [KT18] C. M. Kulkarni, M. Takalikar. „Analysis of REST API Implementation“. In: (2018) (zitiert auf S. 41).
- [LL09] K. B. Laskey, K. Laskey. „Service oriented architecture“. In: *Wiley Interdisciplinary Reviews: Computational Statistics* 1.1 (2009), S. 101–105 (zitiert auf S. 11).
- [LS80] B. P. Lientz, E. B. Swanson. „Software maintenance management“. In: (1980) (zitiert auf S. 29).
- [LS81] B. P. Lientz, E. B. Swanson. „Problems in application software maintenance“. In: *Communications of the ACM* 24.11 (1981), S. 763–769 (zitiert auf S. 30).

- [Mas11] R. Mason. „How rest replaced soap on the web: What it means to you“. In: *InfoQ, Oct 20* (2011), S. 17 (zitiert auf S. 18).
- [Mel10] I. Melzer. *Service-orientierte Architekturen mit Web Services: Konzepte-Standards-Praxis*. Springer-Verlag, 2010 (zitiert auf S. 12, 14, 17, 18).
- [MLM+06] C. M. MacKenzie, K. Laskey, F. McCabe, P. F. Brown, R. Metz, B. A. Hamilton. „Reference model for service oriented architecture 1.0“. In: *OASIS standard 12* (2006), S. 18 (zitiert auf S. 11).
- [Mog01] G. Mogyorodi. „Requirements-based testing: an overview“. In: *Technology of Object-Oriented Languages and Systems, 2001. TOOLS 39. 39th International Conference and Exhibition on*. IEEE. 2001, S. 286–295 (zitiert auf S. 30).
- [MP+13] S. Mumbaikar, P. Padiya et al. „Web services based on soap and rest principles“. In: *International Journal of Scientific and Research Publications* 3.5 (2013), S. 1–4 (zitiert auf S. 24).
- [Mul12] B. Mulloy. „Web API Design-Crafting Interfaces that Developers Love“. In: *URL: <https://pages.apigee.com/rs/apigee/images/api-design-ebook-2012-03.pdf>* (2012) (zitiert auf S. 19).
- [MZM+17] C. Mateos, A. Zunino, S. Misra, D. Anabalon, A. P. Flores. „Managing Web Service Interface Complexity via an OO Metric-based Early Approach“. In: 20 (Dez. 2017), S. 2 (zitiert auf S. 14).
- [NDFCF18] *Design First vs Code First*. <https://www.networknt.com/design/design-first/>. Accessed: 2018-10-01 (zitiert auf S. 26, 27).
- [Nid96] S. R. Nidumolu. „Standardization, requirements uncertainty and software project performance“. In: *Information & Management* 31.3 (1996), S. 135–150 (zitiert auf S. 30).
- [Nordic18] *What is the Richardson Maturity Model?* <https://nordicapis.com/what-is-the-richardson-maturity-model/>. Accessed: 2018-09-28 (zitiert auf S. 21, 22).
- [NS11] R. Nelius, D. Slama. *Enterprise BPM: Erfolgsrezepte für unternehmensweites Prozessmanagement*. dpunkt. verlag, 2011 (zitiert auf S. 11, 13).
- [OAbs18] *Basic Structure*. <https://swagger.io/docs/specification/basic-structure/>. Accessed: 2018-10-01 (zitiert auf S. 29).
- [OAdap18] *OpenAPI-Driven API Design*. <https://swagger.io/blog/api-design/openapi-driven-api-design/>. Accessed: 2018-10-01 (zitiert auf S. 29).
- [OAI17] *Open API Initiative Announces Release of the OpenAPI Spec v3 Implementer’s Draft*. <https://www.openapis.org/blog/2017/03/01/openapi-spec-3-implementers-draft-released>. Accessed: 2018-10-01 (zitiert auf S. 27).
- [OMCZ13] J. L. Ordiales Coscia, C. M. Mateos Diaz, M. P. Crasso, A. O. Zunino Suarez. „Anti-pattern free code-first web services for state-of-the-art Java WSDL generation tools“. In: (2013) (zitiert auf S. 38).
- [Pau14] C. Pautasso. „RESTful web services: principles, patterns, emerging technologies“. In: *Web Services Foundations*. Springer, 2014, S. 31–51 (zitiert auf S. 18, 20, 22).

- [PRF07] M. Perepletchikov, C. Ryan, K. Frampton. „Cohesion metrics for predicting maintainability of service-oriented software“. In: *Quality Software, 2007. QSIC'07. Seventh International Conference on*. IEEE. 2007, S. 328–335 (zitiert auf S. 35, 60).
- [PZL08] C. Pautasso, O. Zimmermann, F. Leymann. „Restful web services vs. big'web services: making the right architectural decision“. In: *Proceedings of the 17th international conference on World Wide Web*. ACM. 2008, S. 805–814 (zitiert auf S. 17, 18, 20, 22).
- [RAR13] L. Richardson, M. Amundsen, S. Ruby. *RESTful Web APIs: Services for a Changing World*. O'Reilly Media, Inc., 2013 (zitiert auf S. 19).
- [Restapi18] *HATEOAS Driven REST APIs*. <https://restfulapi.net/hateoas/>. Accessed: 2018-09-27 (zitiert auf S. 19).
- [RGK90] J. Radatz, A. Geraci, F. Katki. „IEEE standard glossary of software engineering terminology“. In: *IEEE Std 610121990.121990* (1990), S. 3 (zitiert auf S. 31).
- [Rod08] A. Rodriguez. „Restful web services: The basics“. In: *IBM developerWorks 33* (2008) (zitiert auf S. 18).
- [Roy08] *rest-apis-must-be-hypertext-driven*. <https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven>. Accessed: 2018-09-27 (zitiert auf S. 19).
- [RR08] L. Richardson, S. Ruby. *RESTful web services*. O'Reilly Media, Inc., 2008 (zitiert auf S. 18).
- [SCKP08] B. Shim, S. Choue, S. Kim, S. Park. „A design quality model for service-oriented architecture“. In: *2008 15th Asia-Pacific Software Engineering Conference*. IEEE. 2008, S. 403–410 (zitiert auf S. 32).
- [Sdr18] *CrudRepository, JpaRepository, and PagingAndSortingRepository in Spring Data*. <https://www.baeldung.com/spring-data-repositories>. Accessed: 2018-10-04 (zitiert auf S. 52).
- [SM98] M. J. C. Sousa, H. M. Moreira. „A survey on the software maintenance process“. In: *Software Maintenance, 1998. Proceedings., International Conference on*. IEEE. 1998, S. 265–274 (zitiert auf S. 30).
- [Sma18] *What are microservices*. <https://smartbear.com/learn/api-design/what-are-microservices/>. Accessed: 2018-09-16 (zitiert auf S. 15).
- [Sne10] H. M. Sneed. „Measuring web service interfaces“. In: *Web Systems Evolution (WSE), 2010 12th IEEE International Symposium on*. IEEE. 2010, S. 111–115 (zitiert auf S. 39).
- [SP15] T. Senivongse, A. Puapolthep. „A maintainability assessment model for service-oriented systems“. In: *Proceedings of the World Congress on Engineering and Computer Science*. Bd. 1. 2015 (zitiert auf S. 32).
- [Sur16] V. Surwase. „REST API Modeling Languages-A Developer's Perspective“. In: *Int. J. Sci. Technol. Eng* 2.10 (2016), S. 634–637 (zitiert auf S. 24).
- [Swa18] *About Swagger*. <https://swagger.io/about/>. Accessed: 2018-10-01 (zitiert auf S. 24, 25).

- [SwaOAS18] *What Is the Difference Between Swagger and OpenAPI?* <https://smartbear.de/blog/develop/what-is-the-difference-between-swagger-and-openapi/?l=ua>. Accessed: 2018-10-01 (zitiert auf S. 25).
- [SwaPre18] *SmartBear Launches Open API Initiative With Key Industry Leaders Including Google, IBM And Microsoft.* <https://smartbear.de/news/news-releases/smartbear-launches-open-api-initiative-with-key-in/>. Accessed: 2018-10-01 (zitiert auf S. 25).
- [Tec18] *Definition SOA Repository and Registry.* <https://searchmicroservices.techtarget.com>. Accessed: 2018-09-18 (zitiert auf S. 12).
- [Thö15] J. Thönes. „Microservices“. In: *IEEE software* 32.1 (2015), S. 116–116 (zitiert auf S. 14).
- [TPSC02] W.-T. Tsai, R. Paul, W. Song, Z. Cao. „Coyote: An xml-based framework for web services testing“. In: *High Assurance Systems Engineering, 2002. Proceedings. 7th IEEE International Symposium on*. IEEE. 2002, S. 173–174 (zitiert auf S. 37).
- [TS12] G. Tiwari, A. Sharma. „Maintainability Techniques for Software Development Approaches—A Systematic Survey“. In: *Special Issue of International Journal of Computer Applications (0975-8887), ICNICT* (2012) (zitiert auf S. 29).
- [WH98] D. N. Wilson, T. Hall. „Perceptions of software quality: a pilot study“. In: *Software quality journal* 7.1 (1998), S. 67–75 (zitiert auf S. 30).
- [Wol18] E. Wolff. *Microservices: Grundlagen flexibler softwarearchitekturen*. dpunkt. verlag, 2018 (zitiert auf S. 15).
- [WPR10] J. Webber, S. Parastatidis, I. Robinson. *REST in practice: Hypermedia and systems architecture*. Ö'Reilly Media, Inc., 2010 (zitiert auf S. 20).
- [YL94] S. W. Yip, T. Lam. „A software maintenance survey“. In: *Software Engineering Conference, 1994. Proceedings., 1994 First Asia-Pacific*. IEEE. 1994, S. 70–79 (zitiert auf S. 30).

Alle URLs wurden zuletzt am 15. 10. 2018 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift